TK1100
Digital Teknologi
3'dje forelesning

DATAMASKIN-

arkitektur

Læringsmål dekket så langt (1)

- definere begrepene **digital** og **analog**
- definere begrepene **system** og komponent og benytte disse i analyse
- definere datamaskin («computer»).
- definere og eksemplifisere begrepet **abstraksjon** i datateknisk forstand
- definere : Hz, bit, nibble, byte, word
- benytte prefixene: Ki, Mi, Gi, Ti både i desimal (k,M,G,T) og binær forstand

Læringsmål (2)

- forklare hvordan tall, tekst, lyd og bilder kan representeres binært og hvilke muligheter og begrensninger dette gir i forhold til analoge metoder.
- redegjøre for prinsippene bak et posisjonstallsystem og konvertere mellom desimale, binære og hexadesimale heltall
- redegjøre for konsekvensene av at i datamaskiner så operer man alltid med en begrenset, forhåndsgitt presisjon («ord-størrelse»)
- forstå prinsippet bak toerkomplement representasjon av heltall og konvertere mellom positive og negative heltall med en forhåndsgitt presisjon
- beskrive virkemåten til IEEE 754 standarden for koding av flyttall, og når denne er ønskelig/nødvendig å bruke
- beskrive oppbyggingen av ulike kode-tabeller for alfanumeriske tegn: US-ASCII, ISO 8859-1, UNICODE (UTF-8 og UTF-16) og benytte disse til å feilsøke vanlige tegnsett-problemer
- kunne forklare prinsippet bak Huffman-koding/komprimering og utføre enkel ordbok-komprimering
- forklare og vise forskjellen på grafikk-formater basert på raster- og vektor-grafikk.
- benytte en hexeditor til å inspisere og endre innholdet i filer i ulike formater

I dag: Datamaskinarkitektur

1. Boolsk algebra
   - NOT, AND, OR, XOR
   - Sekvensielle og kombinatoriske kretser.

2. InstruksjonSettArkitektur (ISA)
   - Hvilke instruksjoner?
   - Størrelse på ord (antall bit)
   - Adresseringsmoduser
   - Registre
   - Dataformater
   - Hva CPU er egnet til å gjøre.

3. Organisering (mikroarkitektur på CPU)
   - Veiene data flyttes (bussarkitektur)
   - Hvordan data lagres, f.eks. Cache
   - …

4. Systemdesign (Neste gang!)
   - Chipset (buss- og minne-kontrollere)
   - Minneorganisering
   - Virtuelt minne?
   - DMA
   - Hva og hvordan det er koplet sammen

x \ y	$\wedge$ 0	$\wedge$ 1
0	0	0
1	0	1

x \ y	$\vee$ 0	$\vee$ 1
0	0	1
1	1	1

x \ y	$\rightarrow$ 0	$\rightarrow$ 1
0	1	1
1	0	1

x \ y	$\oplus$ 0	$\oplus$ 1
0	0	1
1	1	0

Figure 1. Truth tables

Men FØRST

BOOLSK ALGEBRA

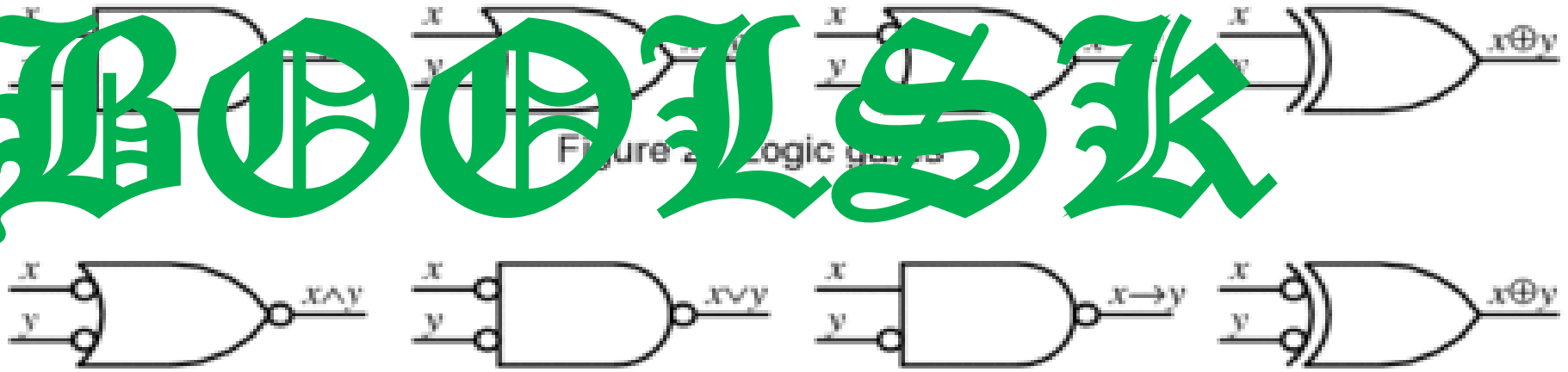
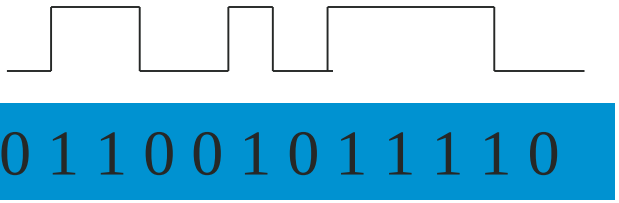


Figure 3. De Morgan equivalents

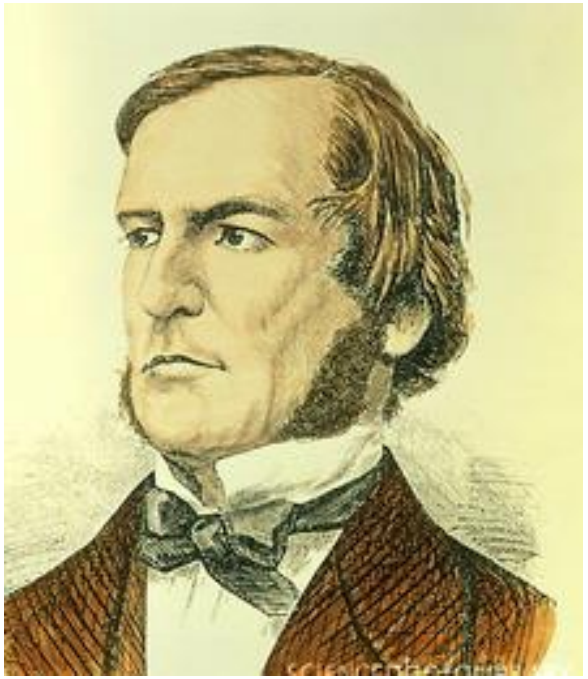
Figure 4. Venn diagrams

Bit og George Boole

- All logikk i **digital** elektronikk er basert på **bit**
- Bruker transistorer som byggeklosser
- Kan være 0 eller 1; høy eller lav spenning
- Logikken som brukes heter **Boolsk Algebra**
  - **George Boole** (1815-1864)

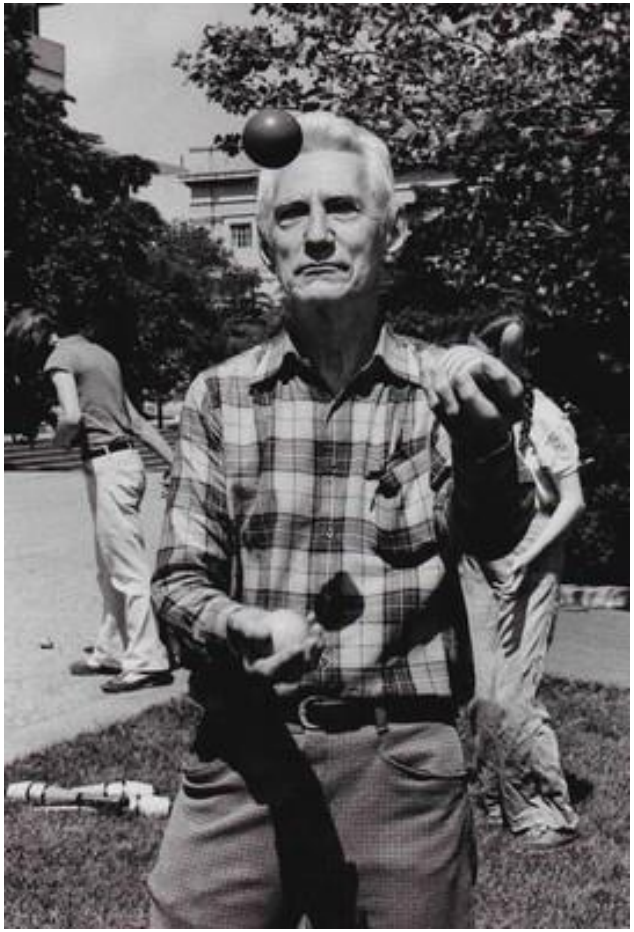


NB! Disse 12 bit'ene er normalt kun et utsnitt av et 32/64 bit ord



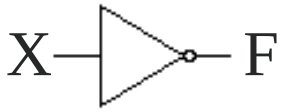
Boolsk algebra

- Består av 3 (4) grunnoperasjoner (porter, gates)
  - IKKE (NOT)
  - OG (AND)
  - ELLER (OR)
  - (EKSKLUSIV ELLER (XOR))
- ”Gjenoppdaget” i 1939 av Claude Elwood Shannon
- Data-elektronikk foretrekker oftest «negativ» logikk
  - NAND, NOR, XNOR



NOT

10010110



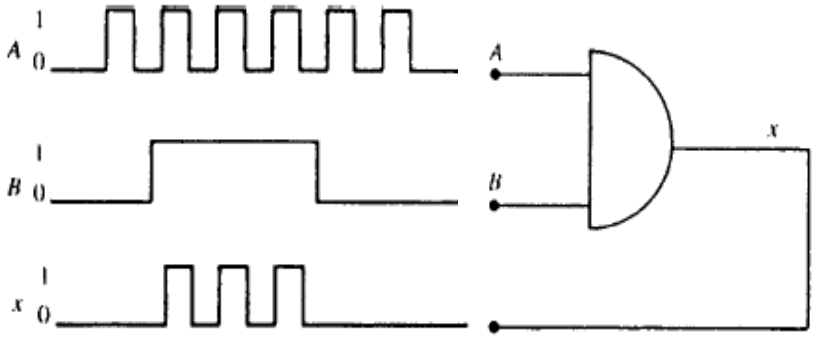
01101001

NOT

Sannhetstabell

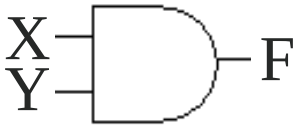
X	F
0	1
1	0

AND og NAND

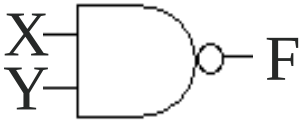


10010101

10101100



10000100



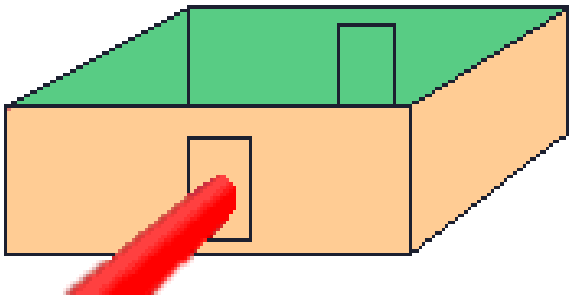
AND

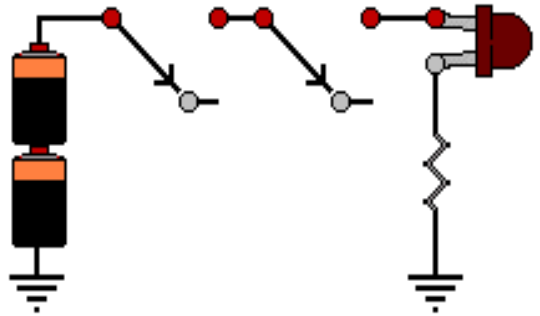
X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

NAND

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

Eksempel: AND



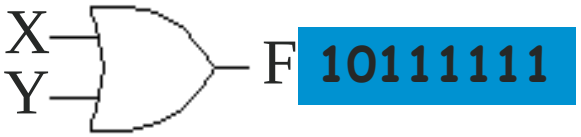


A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

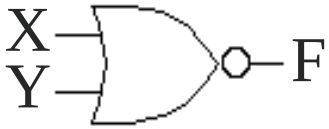
OR og NOR

10111001

10001110



10111111



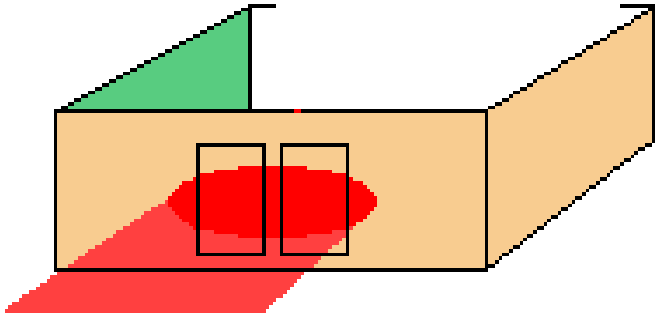
OR

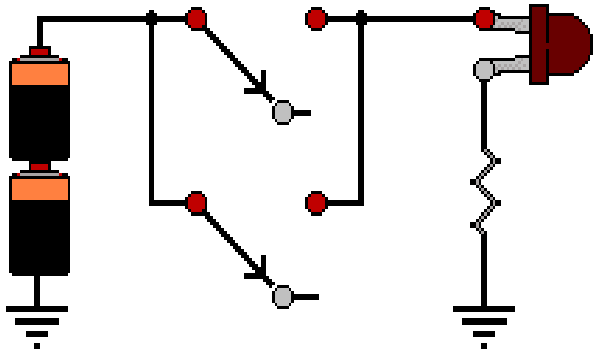
X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

NOR

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0

Eksempel: OR



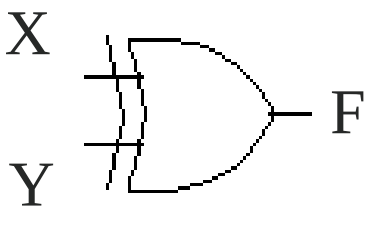


A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

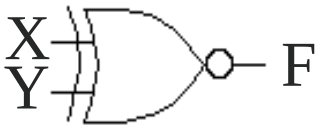
XOR og XNOR

10110010

11001110



01111100



XOR

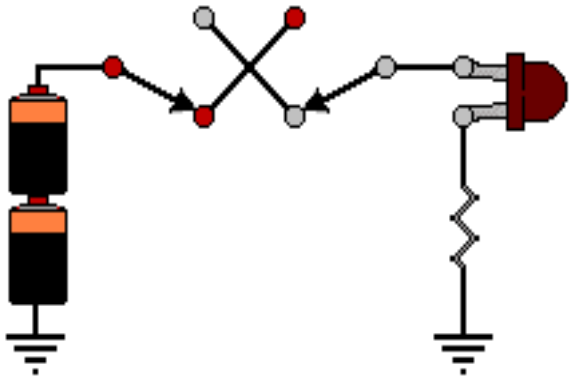
X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

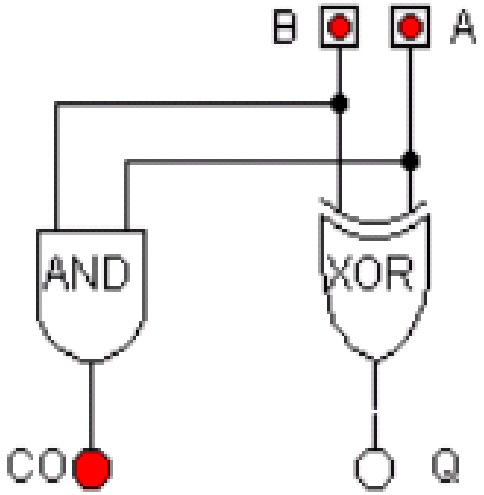
X	Y	F
0	0	1
0	1	0
1	0	0
1	1	1

Eksempel: XOR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0



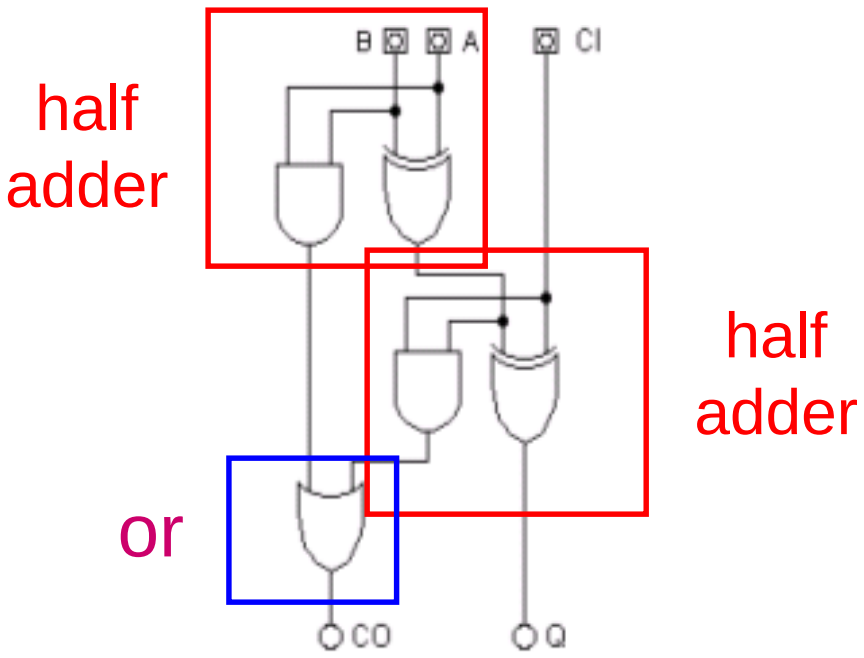
Praktisk eksempel: Half adder



A	B	CO	Q
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Adderer to bit, A og B
- Får svaret i **Q** og menten i **CO**
  - Q = A XOR B
  - CO = A AND B

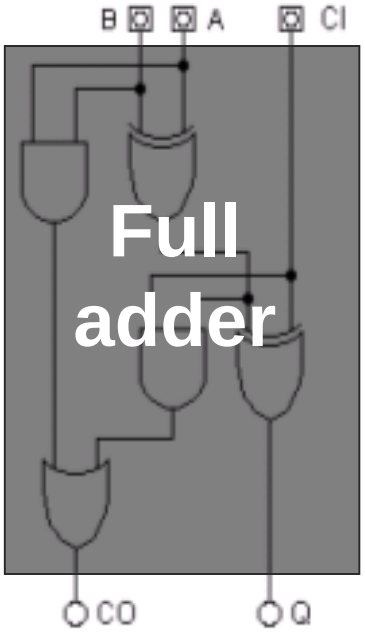
Praktisk eksempel: Full adder



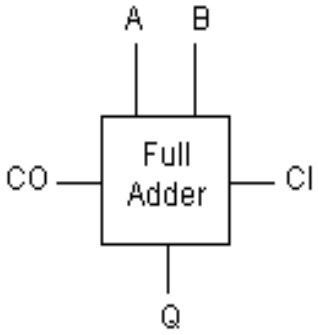
A	B	CI	Q	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Legger sammen to bit, **A** og **B**, og eventuell mente inn på **CI**
- Får svaret i **Q** med menten ut i **CO**

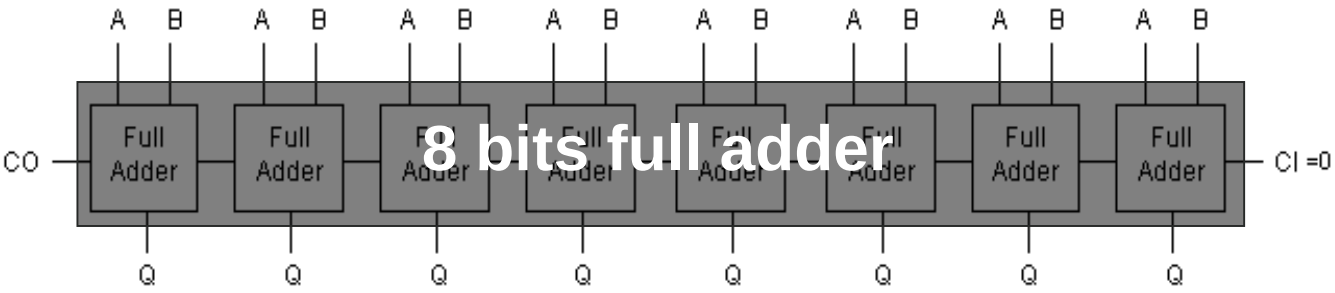
Praktisk eksempel: 8 bits full adder



’svart boks’



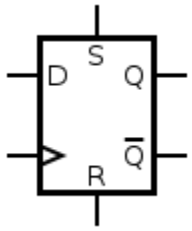
Legger sammen 1 bits tall

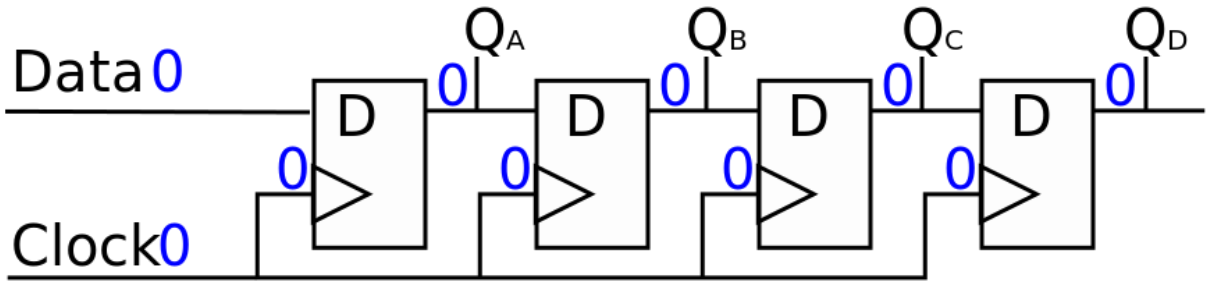


8 fulle addere koplet sammen:
Legger sammen 8 bits tall (kalles en barreladder, og er litt «treig»…)

Sekvensiell vs Kombinatorisk

- Så langt har vi kun sett på det som kalles kombinatorisk logikk
  - **Output** bestemmes utelukkende av **Input** (og tid)
- Sekvensiell logikk har også «hukommelse»
  - Output bestemmes av Input **og** lagrede bits.
  - Brukes bl.a. til å bygge **register** (minne-celler)





Boolean som datatype

- I mange sammenhenger er det ikke samsvar mellom verdiene 0 og 1 og boolske datatyper.
  - Benyttes ofte til utsagnslogikk, der er vi interessert i «true» og «false», ikke 1/0
    - Utsagnet 4==4 har sannhetsverdi true
    - Utsagnet 3==5 har sannhetsverdi false
  - Hvordan «true» og «false» faktisk **kodes binært** er språk-/kompilator-avhengig

- I Java (og andre C-språk) så har man ulike logiske operatorer avhengig om det er sannhetsverdi eller bit-kombinering man vil utføre
  - & er bitwise, && er logisk
    - Dermed blir **3&4 = 0** (0011 & 0100 = 0000),
    - mens **3&&4** gir «true», «syntax error», eller 4…
  - «*Dette kommer det mer av i både PG1100 og DB1100*» har sannhetsverdi **true**

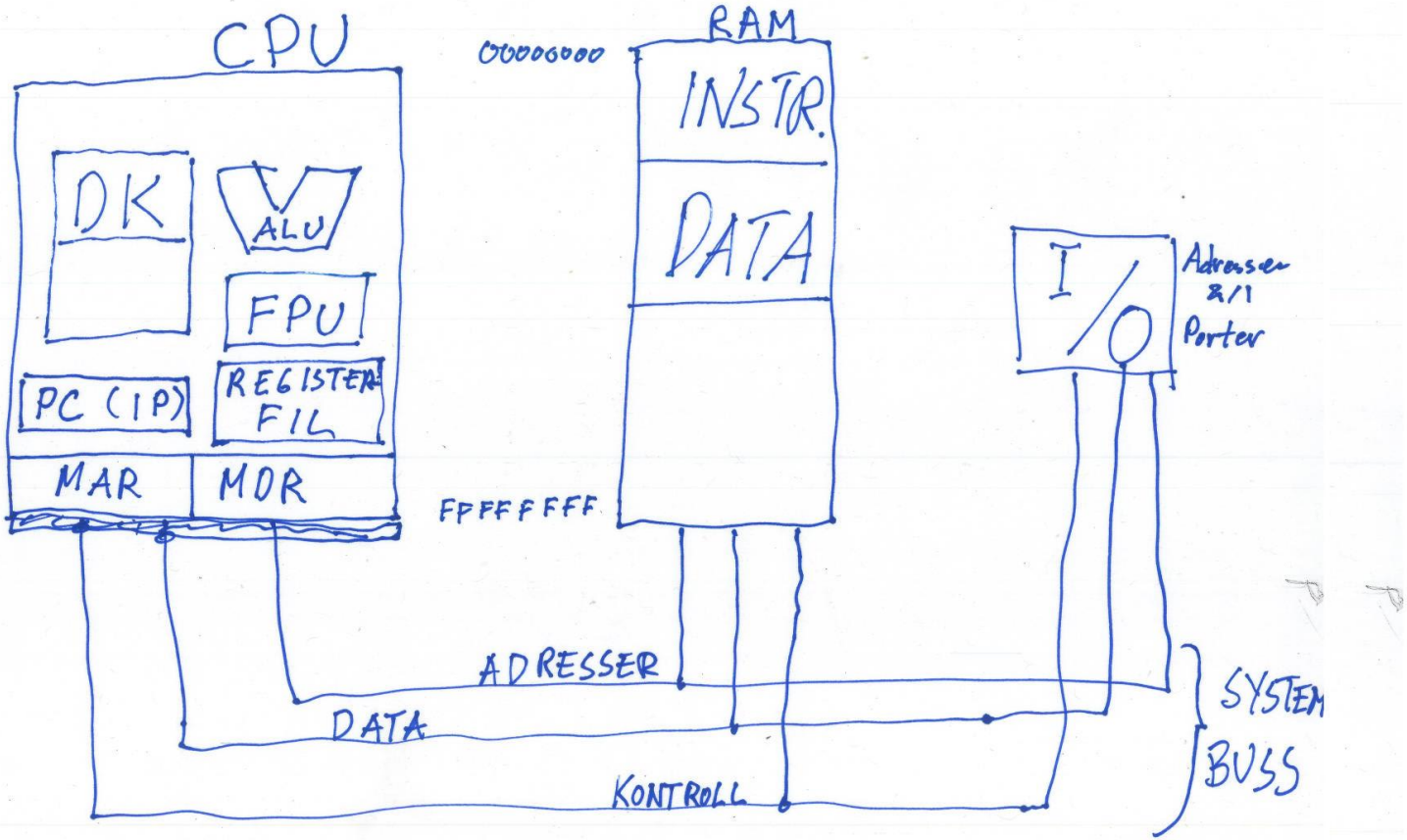
Programmering

VIKTIG

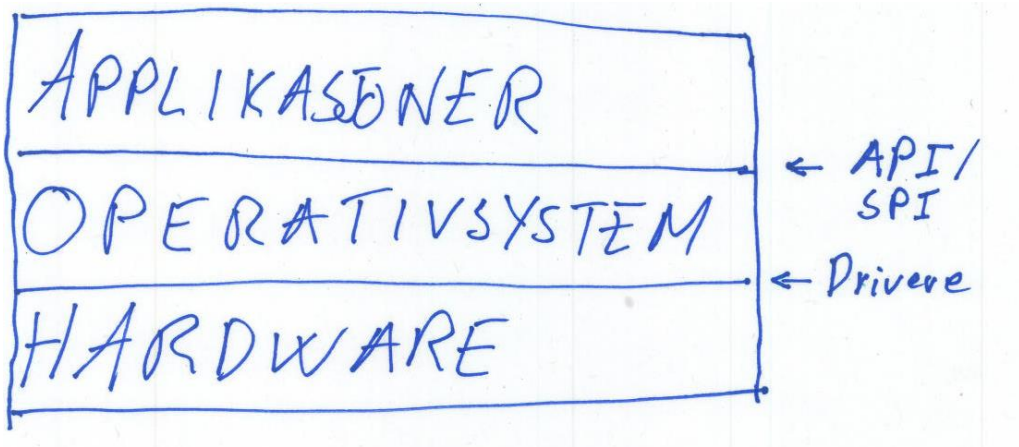
Bitwise AND	skrives	&
Bitwise OR	skrives	|
Bitwise XOR	skrives	^
Bitwise NOT	skrives	~
Logisk NOT	skrives	!
Logisk AND	skrives	&&
Logisk OR	skrives	||

DATAMASKIN ARKITEKTUR

Von Neumann modellen: opened



Software vs Hardware



Hardware

- CPU, RAM, I/O-kontrollere
- Periferibusser
- Tilleggskort
- Harddisk…

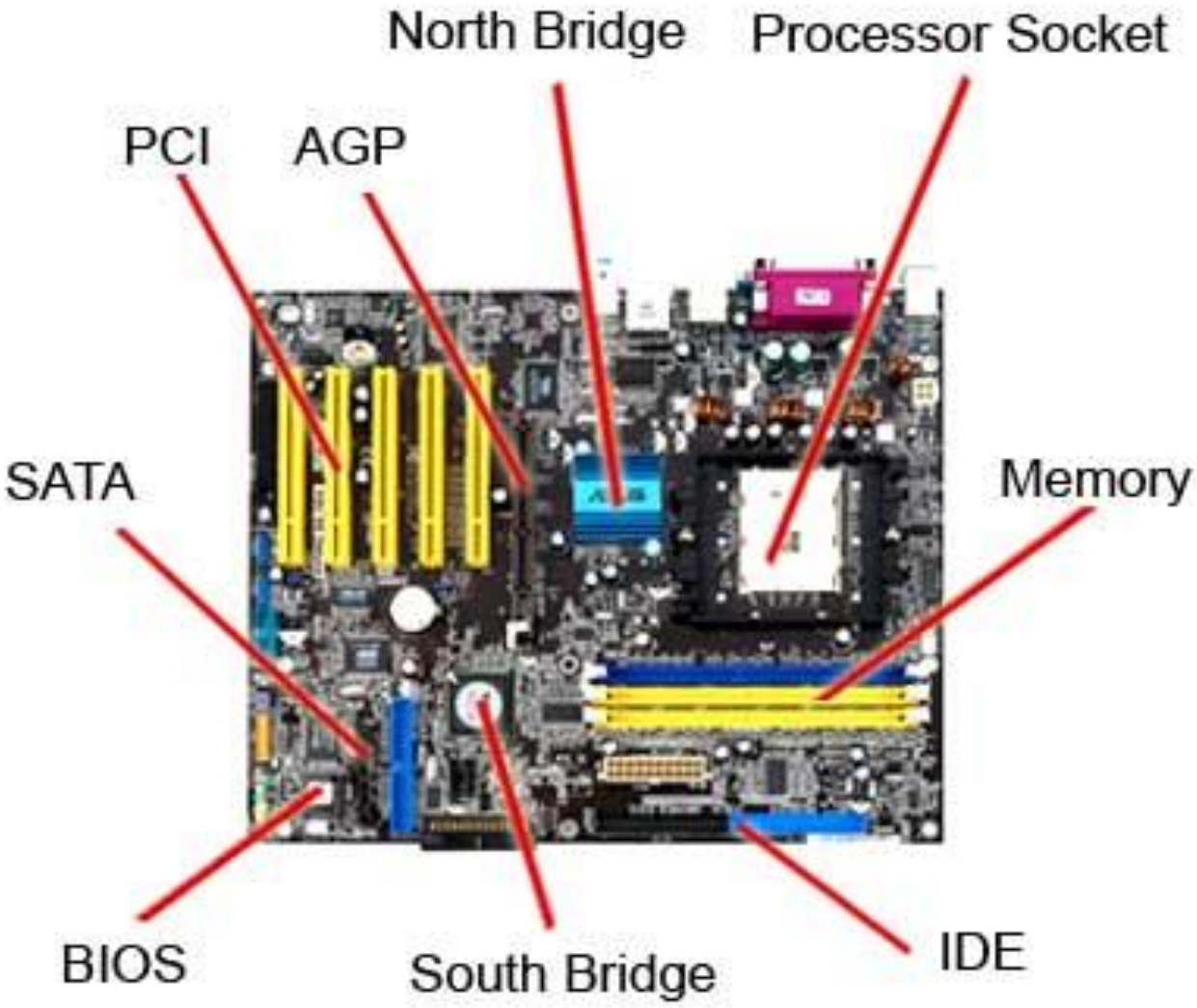
Operativsystem

- Administrer tilgang til CPU, Minne, Utstyr og filer
- Gjør det enklere å bruke hardware
- Abstraksjonslag

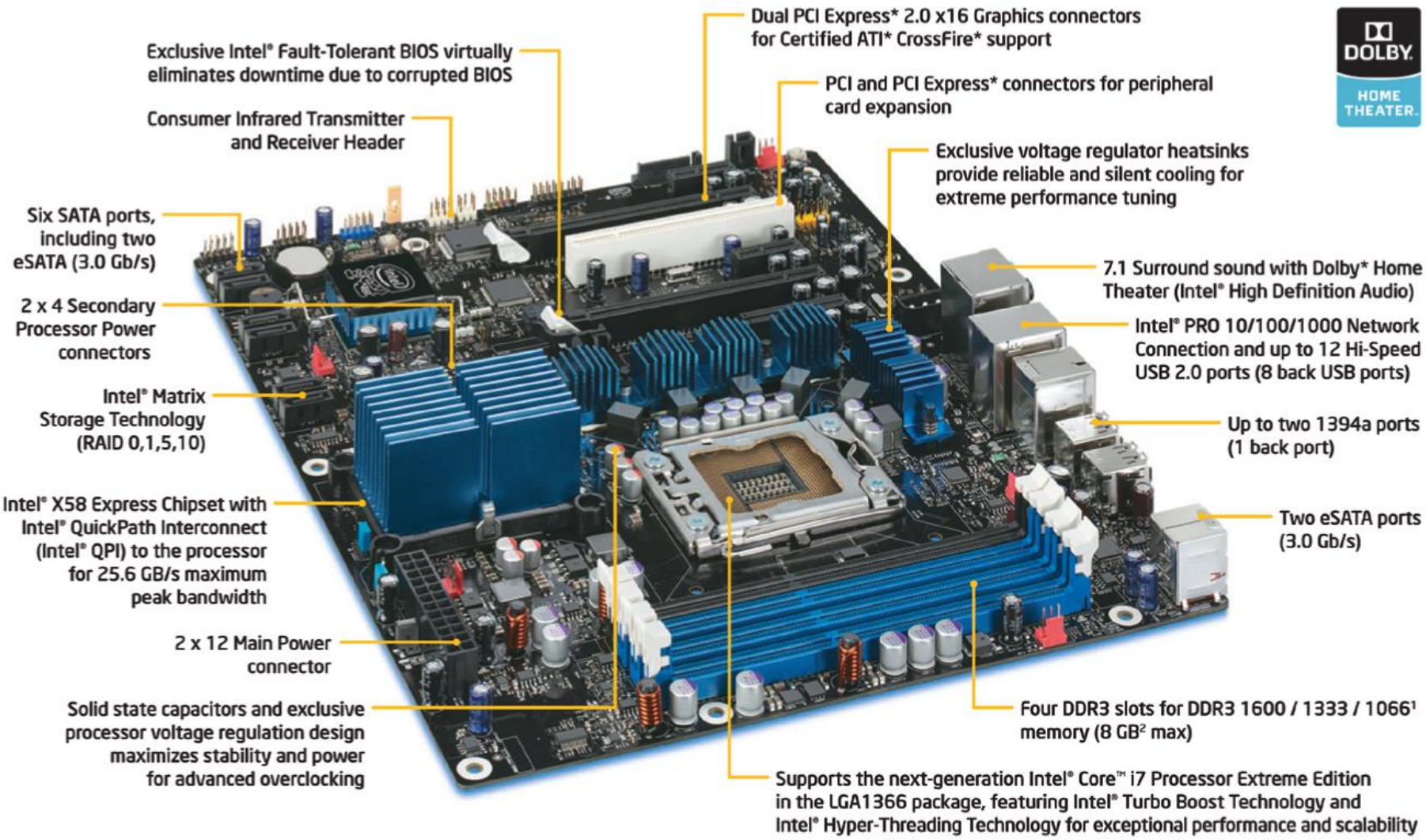
Applikasjoner

- Shell + filbehandler
- Brukerprogrammer
- Kompilatorer

Hovedkort ca 2004 (?)



Hovedkort 2012



- Bruksområder
- Arbeidshastighet (klokkefrekvens)
- Instruksjonsett (ISA)
- Registere
- Arithmetic Logical Unit og Float Point Unit
- Teknikker for å få opp ytelse: pipelining og parallellisering

CENTRAL PROCESSING UNIT

Klassifikasjon

- Bruksområde
  - Embedded <-> Desktop <-> Tungregning
- Klokkefrekvens
- Instruksjonsett (ISA)
  - RISC vs CISC
- **Registerfil**størrelse
- Antall funksjonelle enheter ALU og FPU
- Superskalaritet og parallellitet
- Socket-type

- Energibruk og kjølingsbehov

Produsenter

- Intel og AMD
  - Konstruerer og produserer CPUer, chipset m.m.
  - Fokus: **Desktop**, **server**, mobil, embedded
  - Complex Instruction Set Computing
- MIPS og ARM
  - Konstruerer og lisenserer CPU-kjerner mm til ulike produsenter
  - Fokus: **Embedded**, **mobil**
  - Reduced Instruction Set Computing

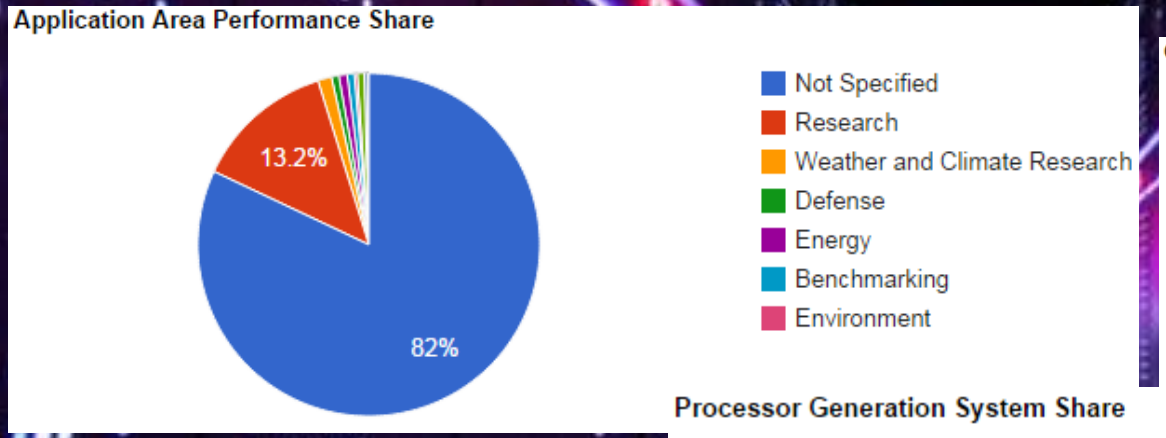
Tungregning

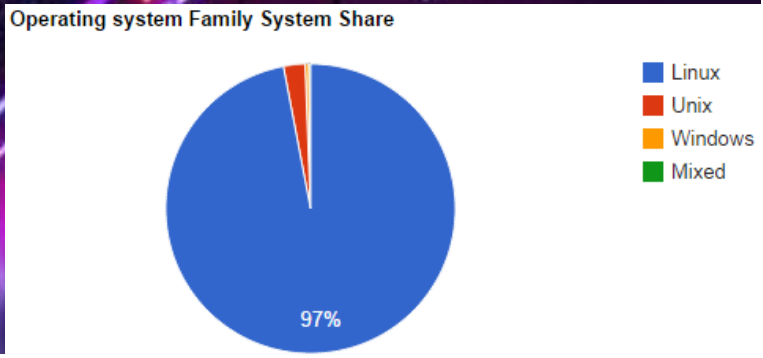
Pr juni 2014 var det ulike Intel Xeon-modeller, som kjører Linux i Clustere dominerer

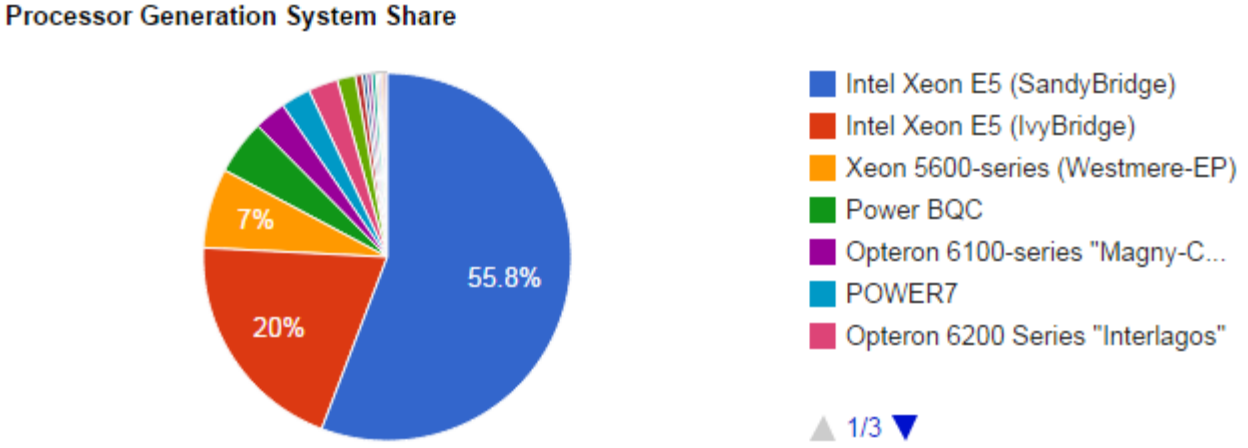
Tianhe-2

- 3.120.000 kjerner
- 33.863 Tflop/s
- 17.808 kW
- 1.024.000 GB RAM
- Kylin Linux

Verdens største og kjappeste...







Kilde: http://top500.org/

Embedded systems

- Laget for bestemte formål
- Styres av en mikroprosessor/-kontroller
- Programvaren er ferdiglaget, ofte ett enkelt program, og ligger gjerne i firmware
  - Kan ofte konfigureres
  - Trenger (ofte) ikke OS
- Eksempler
  - Vaskemaskiner, hjemmeroutere, klokkeradioer, gitarstemmere, bankkort (smartcard), SIM-kort, …

- Par-tre siste årene har Arduino og Raspbury Pi åpnet for mye enklere hjemmeutvikling av ymse embedded systemer

Mobiltelefoner

- Tradisjonelt benytter mobiltelefoner enkle og «billige» CPUer
  - **ARM** eller **MIPS** ISA og kjerne
  - Mobilprodusentene produserer så selv tilpassende versjoner

- Intel har ambisjoner i markedet.
- «Smart-telefoner» ligner (kanskje) mest på en fullverdig PC med eget GSM-kort (med eget «operativsystem»)…

Desktop ≈ x86

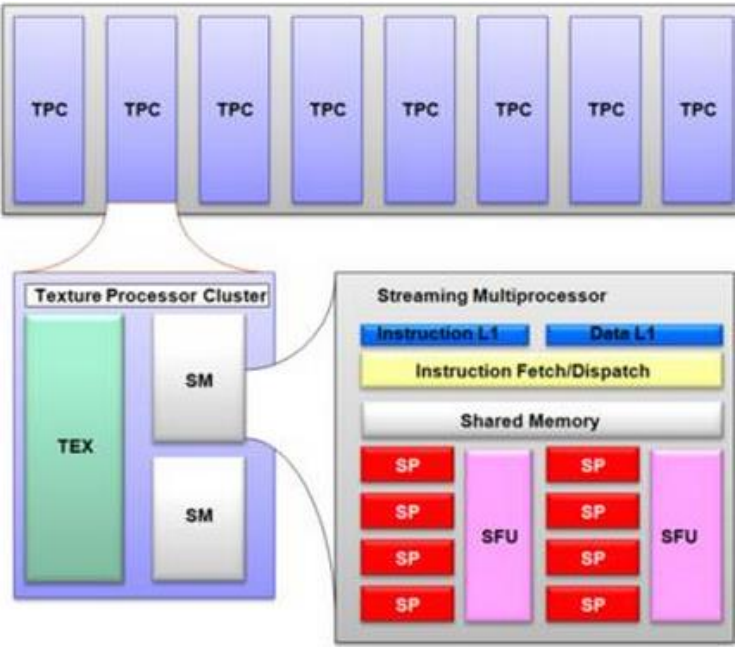
- **x86**
  - x86 er generisk betegnelse på CPUer med samme instruksjonsett som Intel har utviklet
- Apple startet med Motorola 68000 RISC CPU, fortsatte med IBM/Freescale PowerPC CPUer
  - Gikk over til x86 i 2005
- Intel og AMD
  - Videreutvikler fremdeles x86 serien

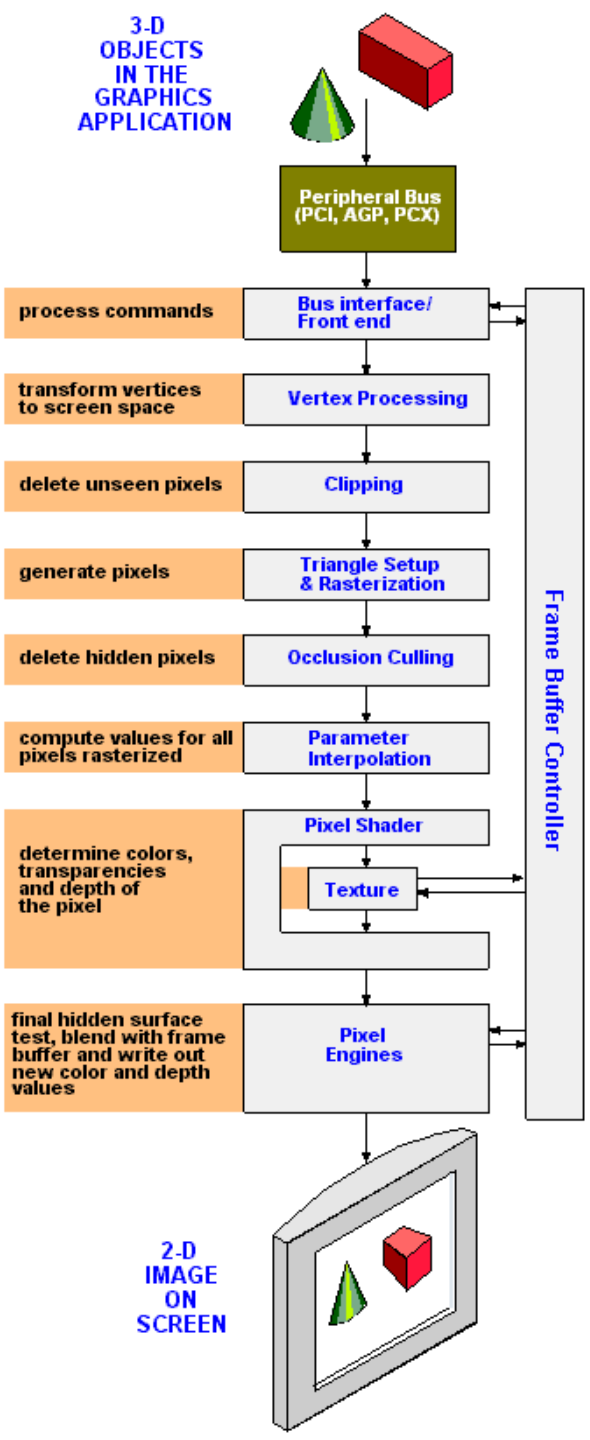
x86 instruksjonsett (CISC)

- Data flytting
  - Flytte data mellom registre og til/fra RAM
  - mov, push, pop, …
- Kontrolloverføring:
  - Utføre betingelsessetninger (if, while) og metodekall
  - je, jg, jmp, call, ret
- Aritmetikk/Logisk:
  - Utføre operasjoner på data i registre
  - cmp, add, sub, inc, mul, imul, not, and, or, xor
- Input/Output: in, out
  - Skrive/lese til porter (og minneadresser)
- Debug og Interrupt håndtering:
  - int, sti, hlt, nop
- **Flyttall** har egne instruksjoner
- SIMD vektor og matriseinstruksjoner
  - MMX, 3D Now!, SSE 1-4
- Egne instruksjoner for å endre prosessortilstand (system)

Graphics Processing Unit

- Enten integrert på hovedkort, eller sitter på grafikkort
- Massivt parallell
  - (nå) ~100vis av parallelle regneenheter
  - Regner på matriser (tabeller) av flyttall
  - Transformerer matematiske modeller til skjermbilder (pixler i framebuffer)





LA OSS DESIGNE EN ENKEL DATAMASKIN!

- ENKEL!!! (Forstå prinsippene)
- Primitiv
- Vi lager ikke elektronikken
- Maskinen kan likevel gjøre noen ting
- Nærmere beskrivelse finner dere i **Kompendium 1**

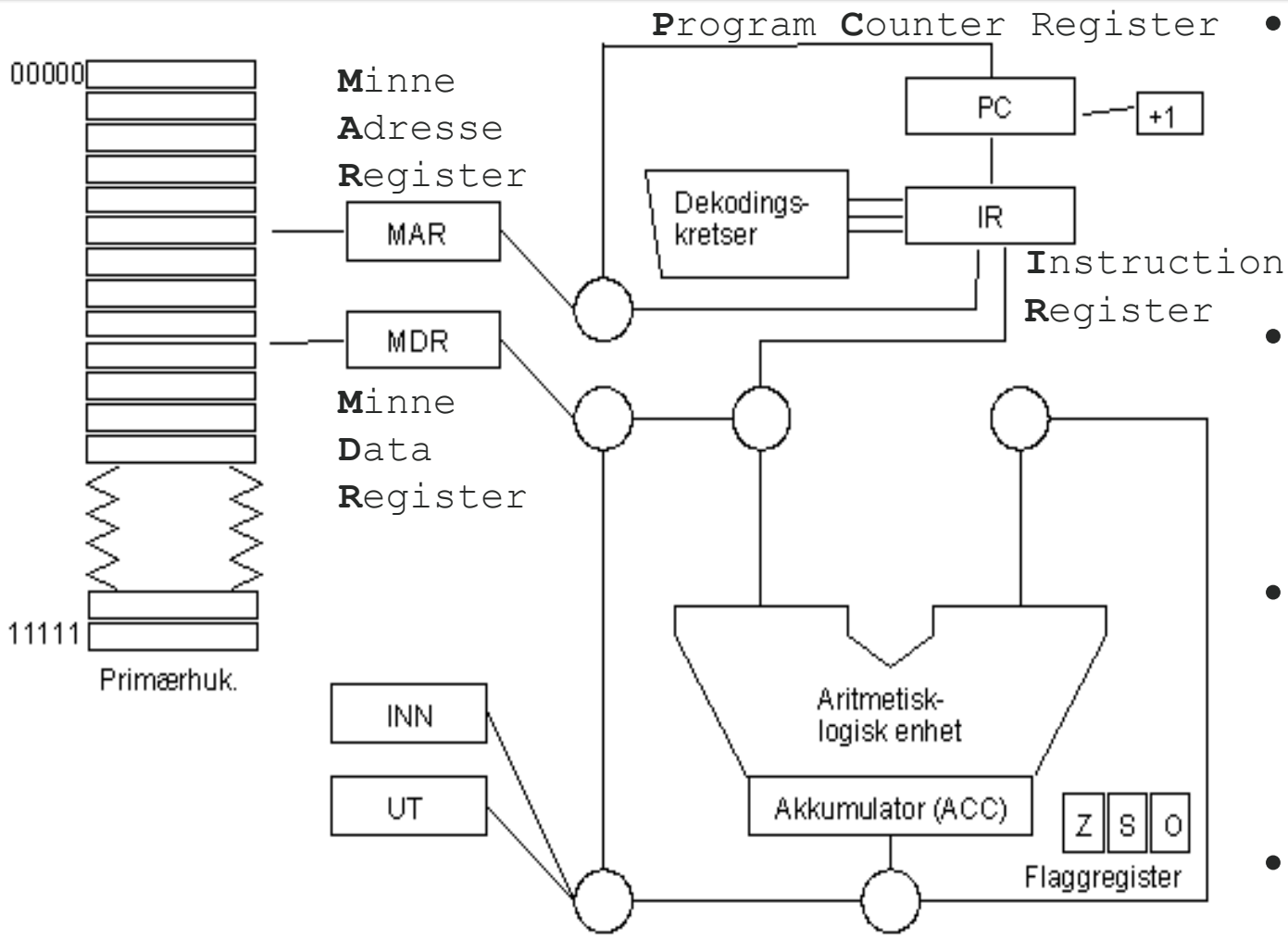
Hva var arkitektur?

- Hvordan bygger vi opp de *instruksjonene* som computeren skal utføre (eksekvere)
  - Hvilke typer instruksjoner skal vi ha?
  - Hvordan skal adressering foregå
- Når vi har bestemt instruksjonssett (**I**nstruction **S**et **A**rchitecture) kan vi designe en CPU
- I praksis blir dette en vekselvirkning mellom maskin-design og ISA-design
  - Vi har ett transistor-budsjett

Register

- Alle bits på CPU må legges i et **register**
  - Et register er adresserbart minne (SRAM) som ligge inne på CPU
- I Assembly-språk har typisk registrene egne «navn» og roller
  - Registrene som brukes til å manipulere variabler og verdier omtales som **arbeidsregistre**
- F.eks. i X86-64
  - **RIP** er registeret som inneholder adressen til hvor i minnet neste instruksjon skal hentes fra
  - **RAX** er registeret som oftest brukes til å lagre resultater i (akkumulator)
  - **RCX** brukes oftest til nedtelling (tenk for-løkke)
  - + mange flere (minnebehandling, kjøring,…)

PrimRisc v. 0.1 layout



- Akkumulator-basert
  - Kun **ett** arbeids-register
- 8 bit adressering
  - Bruker bare 5
- Load/store
  - Alt går via RAM
  - Enkelt
- Kan «egentlig» bare addere

Register-«filen»

- **PC** (Programteller).
  - Dette registeret vil til enhver tid inneholde adressen til den lokasjon i primærlageret som neste instruksjon skal hentes fra.
- **IR** (Instruksjonsregister).
  - Dette registeret oppbevarer instruksjoner etter at de har blitt hentet fra primærlageret og før de sendes over i dekoderkretsene.
- **MAR** (adresseregister).
  - Når vi ønsker å lese noe fra, eller legge noe inn i primærlageret må vi først plassere adressen i MAR.
- **MDR** (dataregister).
  - Når vi gjør en leseoperasjon fra primærlageret, plasseres resultatet her. Skal vi skrive til primærlageret, må vi først plassere det vi skal skrive i MDR
- **ACC** (Accumulator).
  - Dette registret brukes først og fremst i forbindelse med regneoperasjoner. Det er også mulig å lese innholdet av en minneadresse inn i akkumulatoren med instruksjonen LOAD, og lagre innholdet av akkumulatoren med instruksjon STORE.
- FLAGG-registeret
  - Trenger vi i forhold til å kunne gjøre betingede hopp (if, while og lignende)
  - Z-flagget (Zero-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir null.
  - S-flagget (Sign-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir negativt.
  - O-flagget (Overflow-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir for stort (overflyt) for å få plass i ACC-registeret

Instruksjoner

Mnemonic	Opkode	Beskrivelse	Funksjon
STOP	0000	Stopp maskinen	
LOAD	0001	Les inn i akkumulator	(ACC) ← (MDR)
STORE	0010	Lagre akkumulator	(MDR) ← (ACC)
ADD	0011	Adder til akkumulator	(ACC) ← (ACC) + (MDR)
SUB	0100	Subtraher fra akkumulator	(ACC) ← (ACC) - (MDR)
BRANCH	0101	Hopp uansett	(PC) ← IR<4:0>
BRZERO	0110	Hopp hvis resultat = 0	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
BRNEG	0111	Hopp hvis resultat < 0	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
BROVFL	1000	Hopp hvis overflyt	(PC) ← IR<4:0> eller (PC) ← (PC) + 1
IN	1001	Les inn fra tastatur	(ACC) ← INN
OUT	1010	Skriv til skjerm	UT ← (ACC)

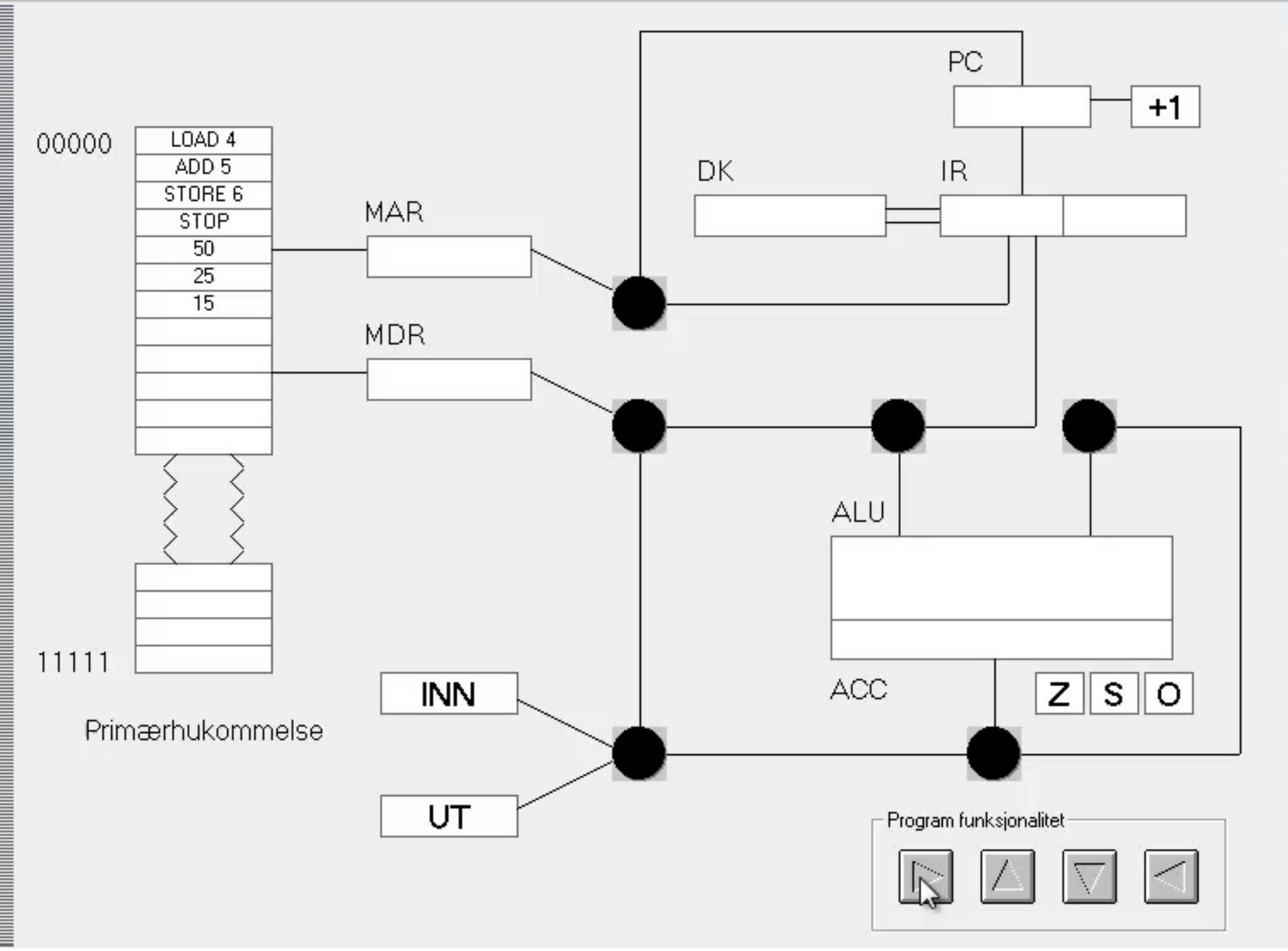
- Alle instruksjoner er 16 bit (2 byte)
  - 1 byte til **opkode**
    - Bruker bare 4 bit (i denne versjonen)
  - 1 byte i instruksjonen til adresser
    - Bruker bare 5 bit (i denne versjonen)

Eksempel på et program i minnet

Minneadresse binært/(desimalt)	Maskinkode (opkode+adresse) binært/(desimalt)	Asembly-instruksjon
00000 (0)	0001 0000 0000 0100 (1)(4)	LOAD 4
00001 (1)	0011 0000 0000 0101 (3)(5)	ADD 5
00010 (2)	0010 0000 0000 0110 (2)(6)	STORE 6
00011 (3)	0011 0000 0000 0000 (0)(-)	STOP
00100 (4)	0000 0000 0000 0001 (0)(1)	
00101 (5)	0000 0000 0000 0010 (-)(2)	
00110 (6)	0000 0000 0000 0000 (-)(0)	

```
/* Last inn i akkumulatoren data fra adresse 4,
adder med data fra adresse 5,
lagre på adressen som ligger på adresse 6,
Avslutt kjøringen */
```

- *Kan* bruke en simulator av denne arkitekturen (buggy) på dagens øving
  (Se http://youtu.be/igthwn_qGVI)



Primærhukommelse

Analyse/Kritikk av PrimRISC 1

- Grei til å demonstrere en Fetch-Execute syklus og hvordan instruksjoner *kan* være bygd opp
- Mangler Pipelining
- Fattig instruksjonssett
  - 8 mot x86-64 sine ~1000
- Støtter ikke Systemprogrammering (OS), multitasking eller virtuelt minne
- «Ganske klein, men dog så enkel»

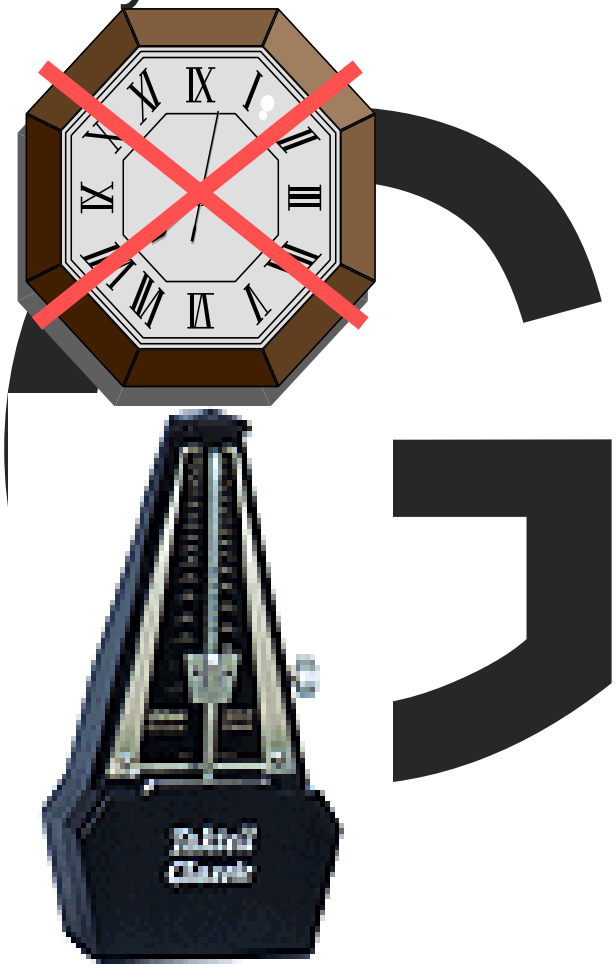
Kritikk 2

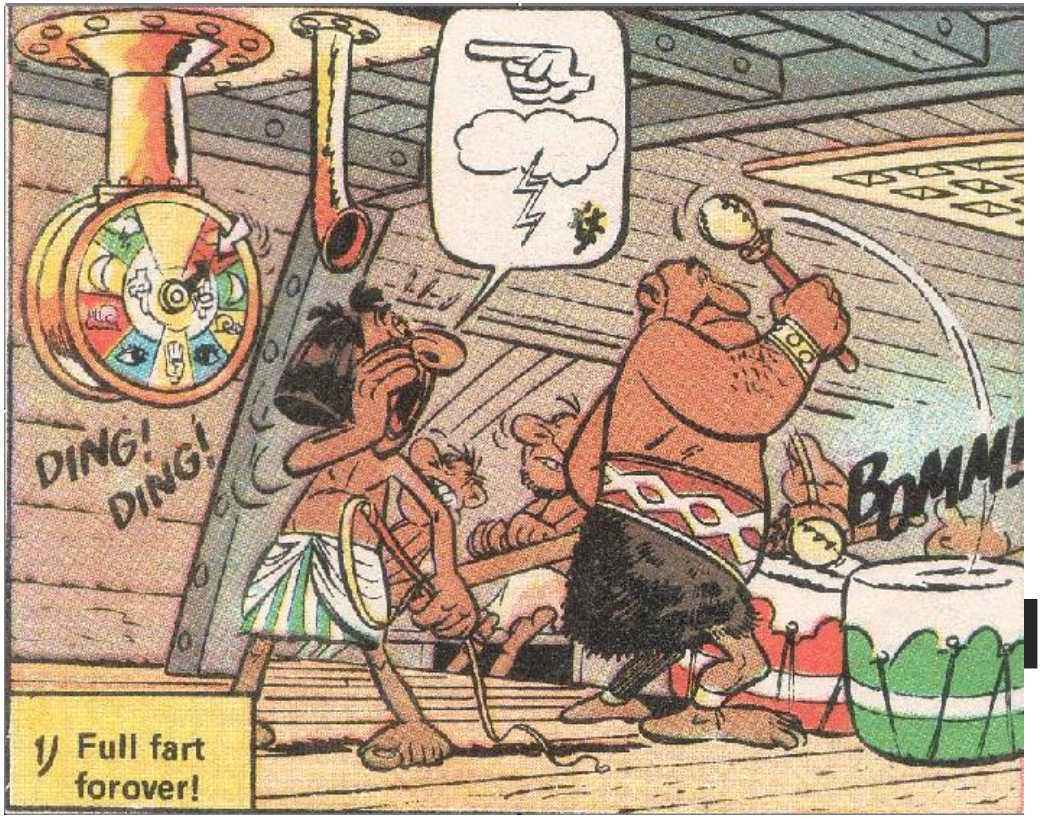
- **Har** støtte for betingede hopp
  - Kan implementere `if`, `for`, `while` o.l.
- Mangler (skikkelig) HW-støtte for metoder!!!
  - I Intel/AMD løses dette med egne registre (`SS` og `SP`) og instruksjoner (`call`) som støtter bruk av et eget minneområde som **stack**.
- Har ikke støtte for **interupt**!
  - Mer om dette de to neste gangene..

TEKNIKKER OG BEGREPER

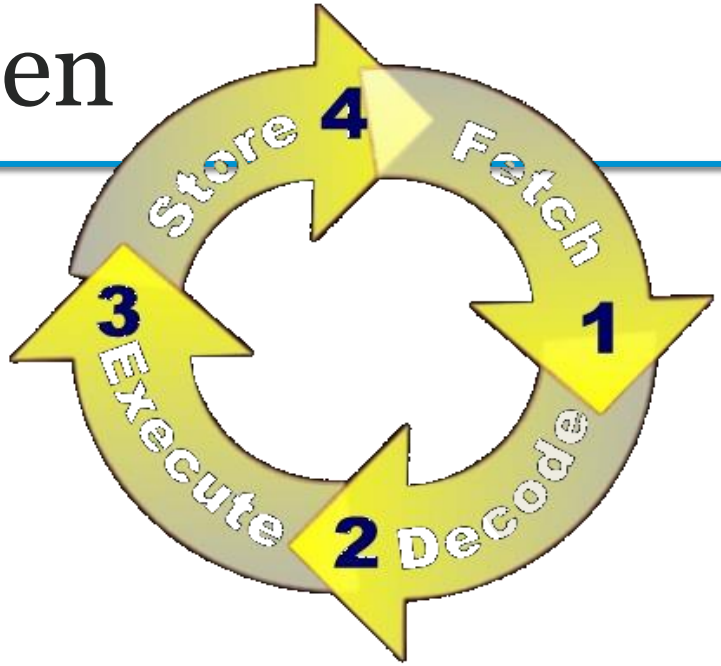
System-klokken (Hz)

- System-klokken holder takten, viser ikke tiden!





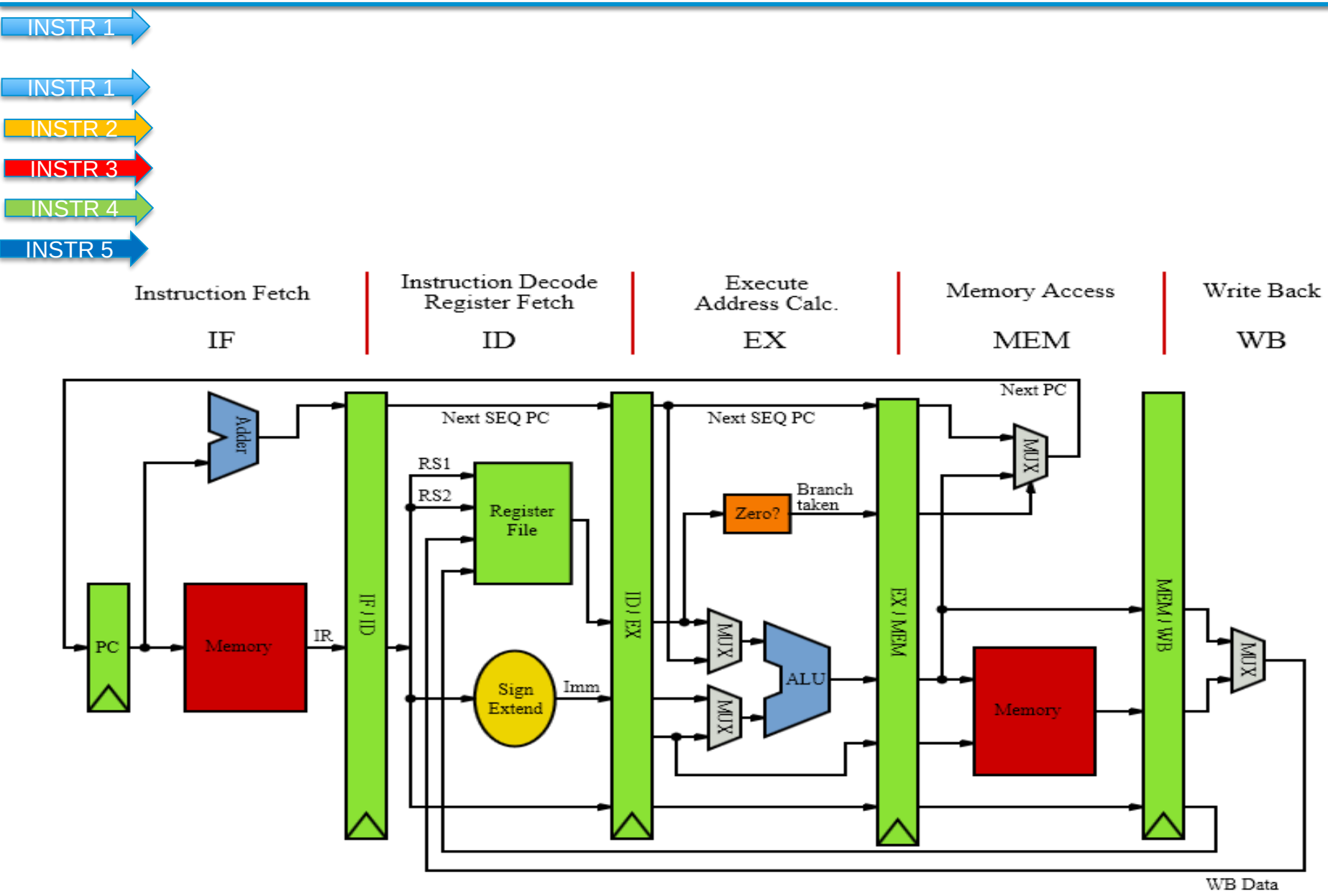
Fetch-Execute-syklusen



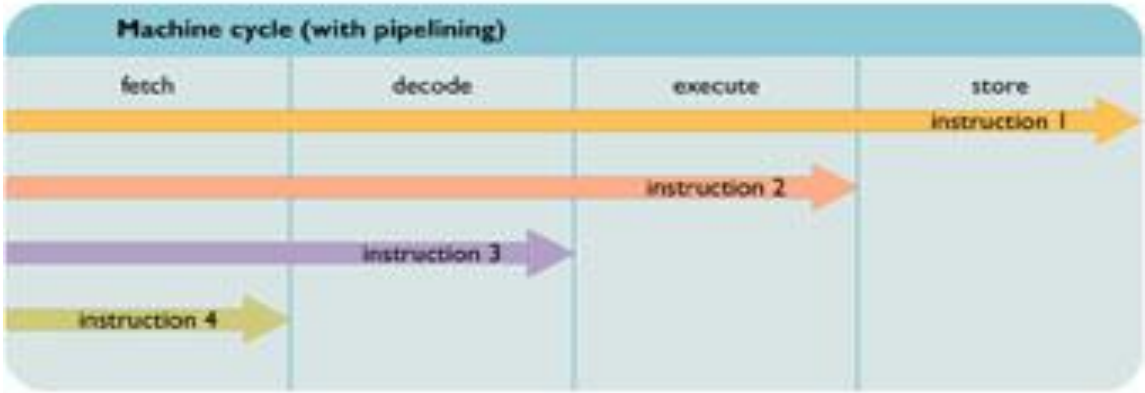
1. **Fetch**: hente instruksjon fra minne og inn i register
2. **Decode**: Tolke instruksjonen og sette opp kretsene som utfører den
3. **Execute**: Utføre instruksjonen på data/registre (endre tilstand)
4. **Store**: Lagre resultat (tilbake i minnet)

Instruction Fetch (IF)
Instruction Decode (ID)
Data Fetch (DF)
Instruction Execution (EX)
Result Return (RR)

Ex: MIPS


INSTR 1
INSTR 1
INSTR 2
INSTR 3
INSTR 4
INSTR 5
Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
RS1
RS2
Register
File
Zero?
Branch
taken
MUX
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
Sign
Extend
Imm
ALU
Memory
WB Data

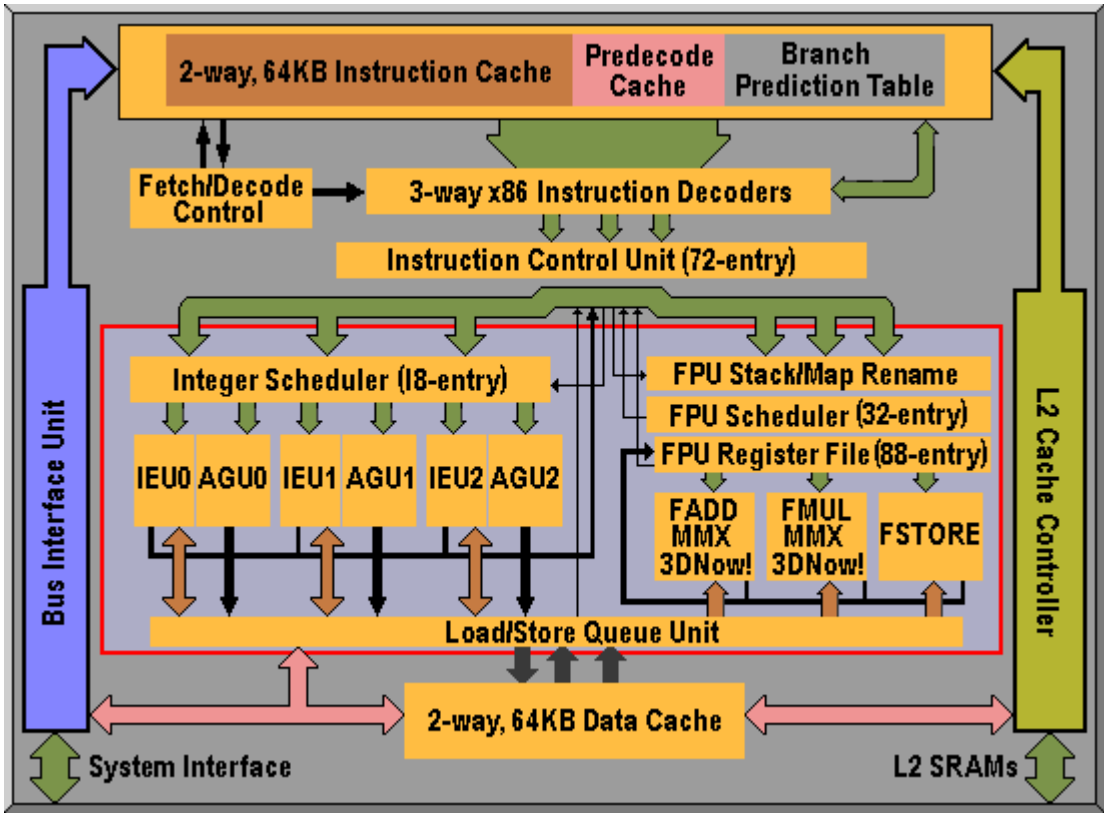
Pipelining



- Siden det er forskjellige enheter på CPU som henter, dekoder, utfører og lagrer kan disse kjøres i parallell («samlebånd»)
  - Kan starte å hente en ny instruksjon så fort denne er overlatt til dekoding
- I moderne Intel/AMD CPUer det «dype og lange» pipelines..
  - 15-25 trinn ->

Superskalaritet

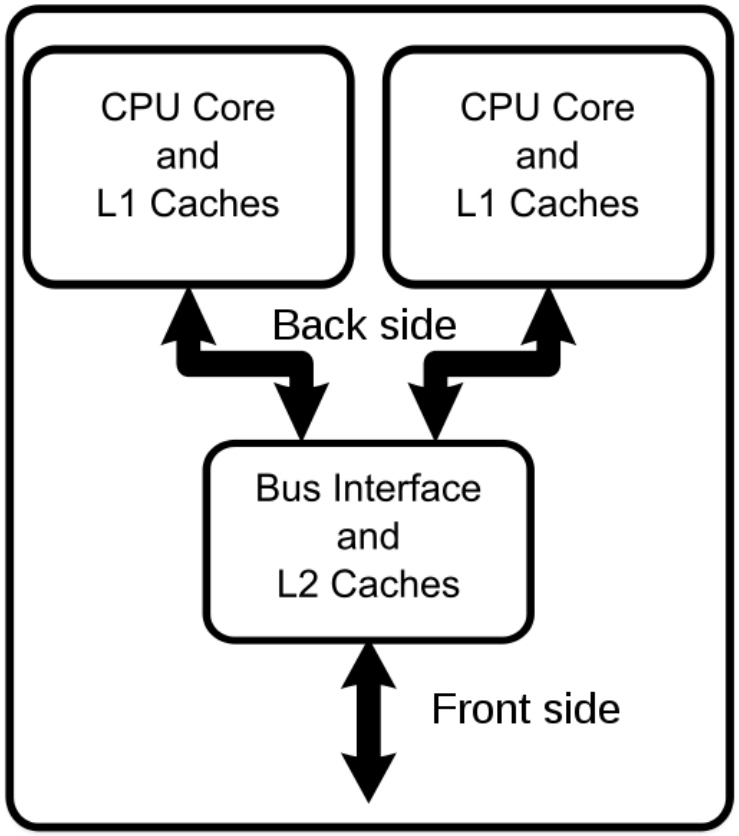
- Superskalaritet er en form for parallellisering der vi har flere (parallelle) **ALU** og **FPU** tilgjengelig på CPU
- Kan da utføre (enda) flere instruksjoner samtidig!



AMD Athlon hadde tre ALUer og en SIMD flyttallenhet

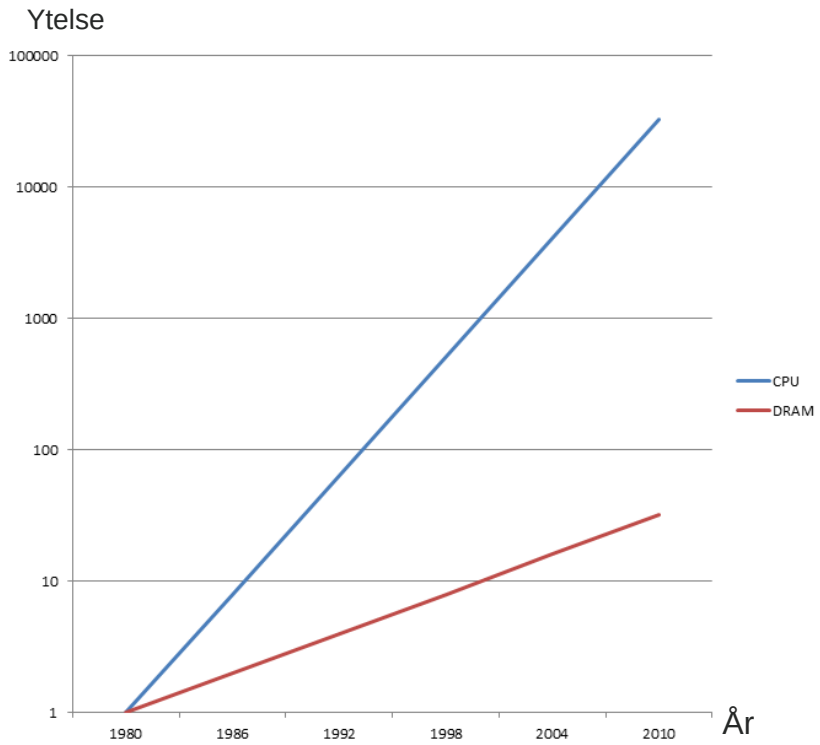
Flere «kjerner»

- CPU inneholder flere fullverdige «under-CPUer», som kan kjøre programvare parallellt.
- For at dette skal være vits i må man ha programmer som **kan** parallelliseres og ikke må kjøres rent sekvensielt!



CPU vs DRAM

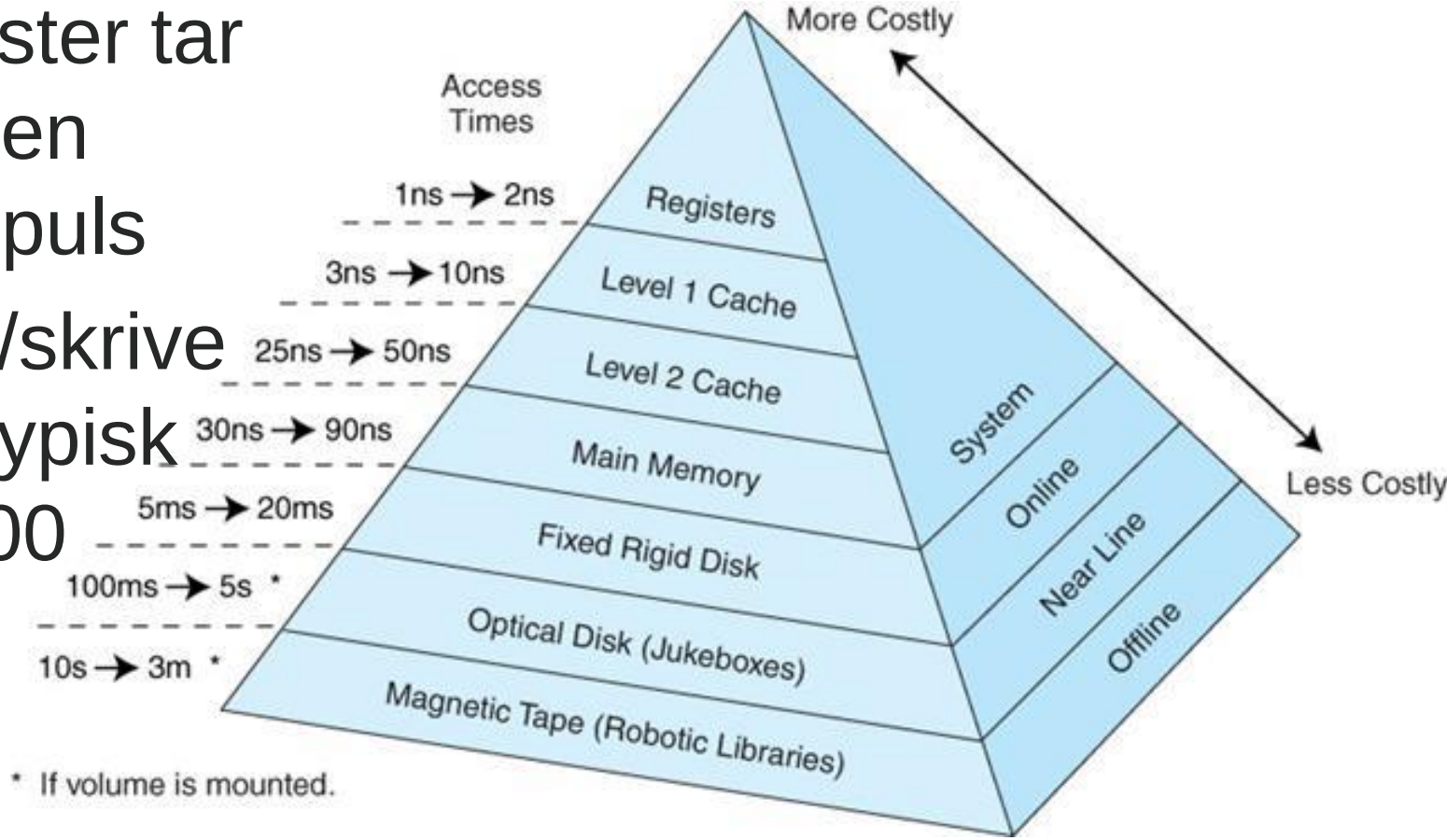
- CPU ytelse/hastighet ha doblet seg annet hvert år
- DRAM ytelse/hastighet hvert sjette år!



- Det fullstendige programmet ligger i RAM
  - Det er bare enkeltinstruksjoner og enkeltdata som er på CPU
- ->PROBLEM!!!
- Utfordringen er å holde pipelinen full

Minne-hierarkiet

- Å gjøre noe i et register tar typisk en klokkepuls
- Å lese/skrive RAM typisk 150-300

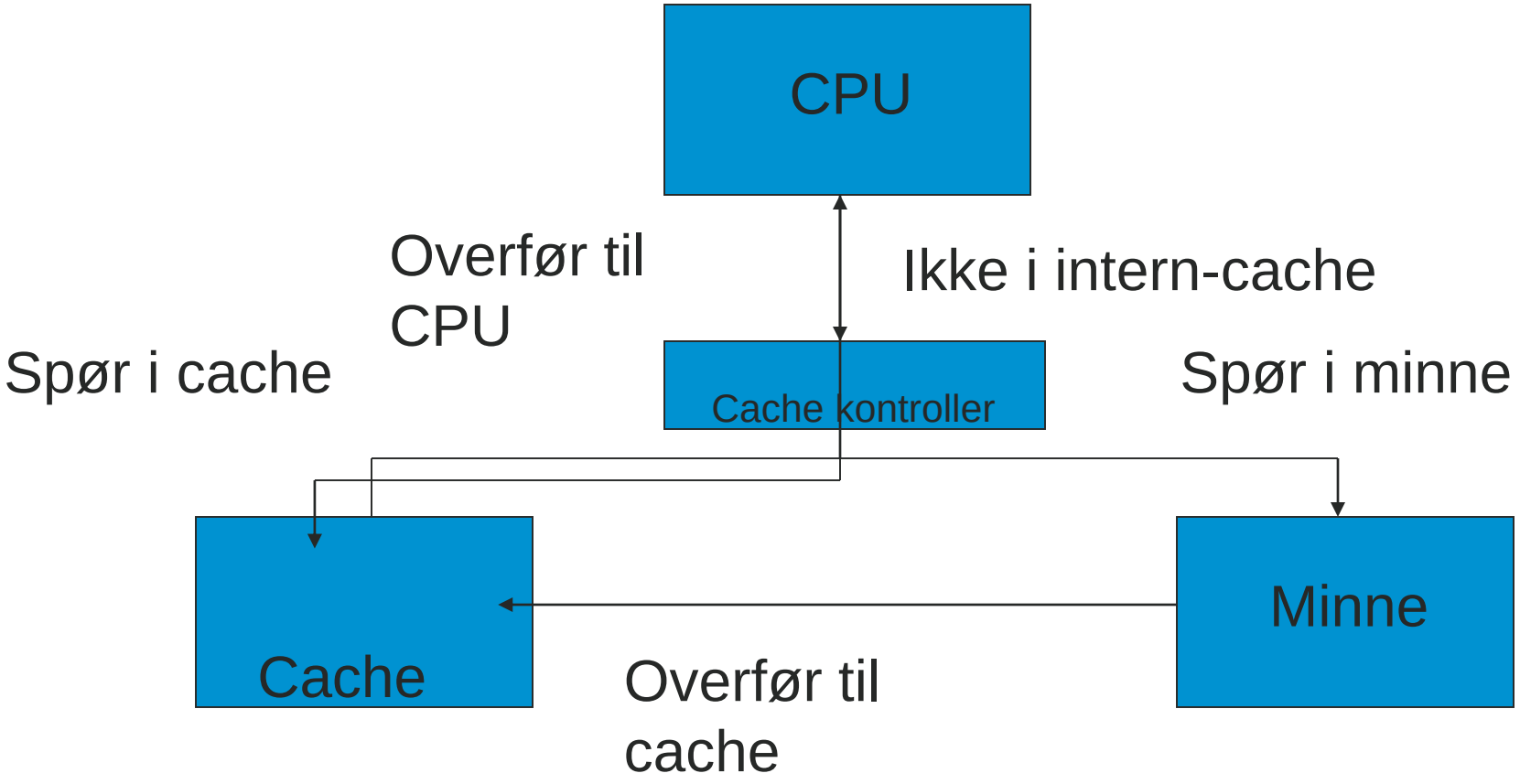


Cache

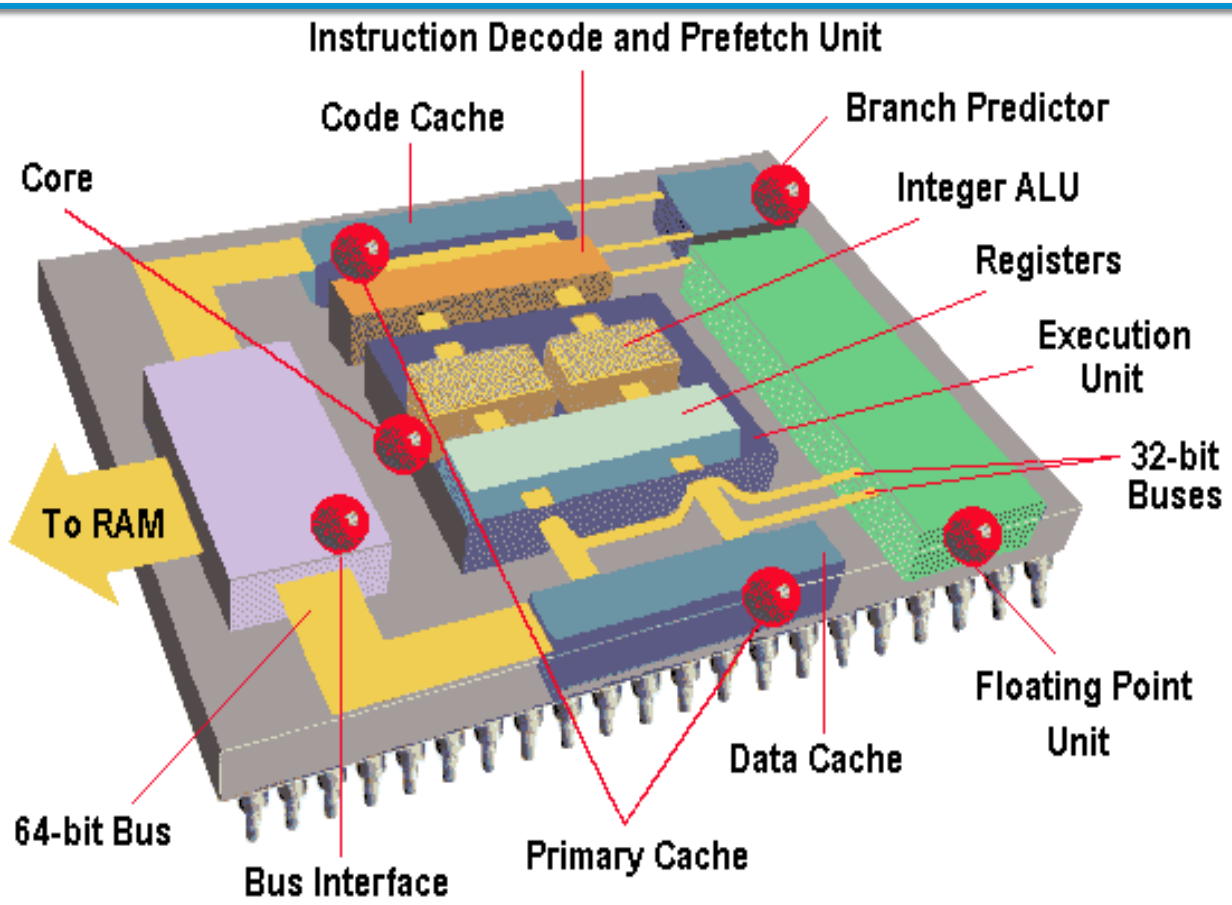
- Lokalitetsprinsippet
  - Ca 5 % av koden kjøres ca 95% av tiden
  - Instruksjoner som skal utføres er lagret i den rekkefølgen de skal utføres
- Cacheing benytter seg av dette ved at vi mellomlagrer sist innleste instruksjoner i cache-minne
  - Mye løkker medfører lett tilgjengelighet…
- + Pageing
  - Vi leser typisk ikke inn enkeltinstruksjoner til Cache, men minneblokker på 4 KiB

Cache

- I prinsippet



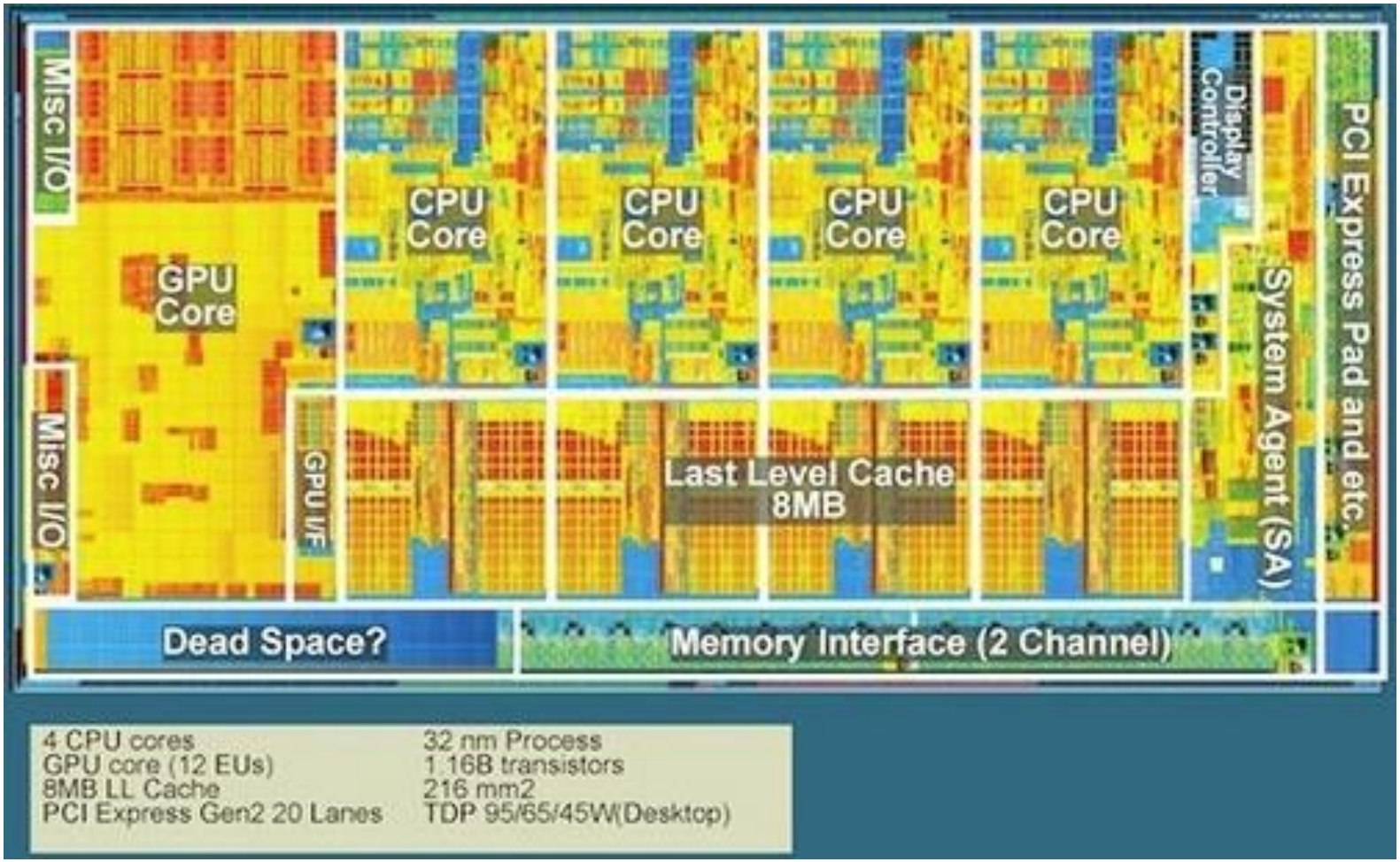
Pentium (IA32) layout (forenklet)



- Branch Prediction
  - «Gjetter» hvilke instruksjoner som skal kjøres etterpå og henter dem inn på CPU
- Cacheing
  - Instruksjoner i egen cache (skal aldri tilbake til RAM!)
  - Data i egen cache

Sandy Bridge

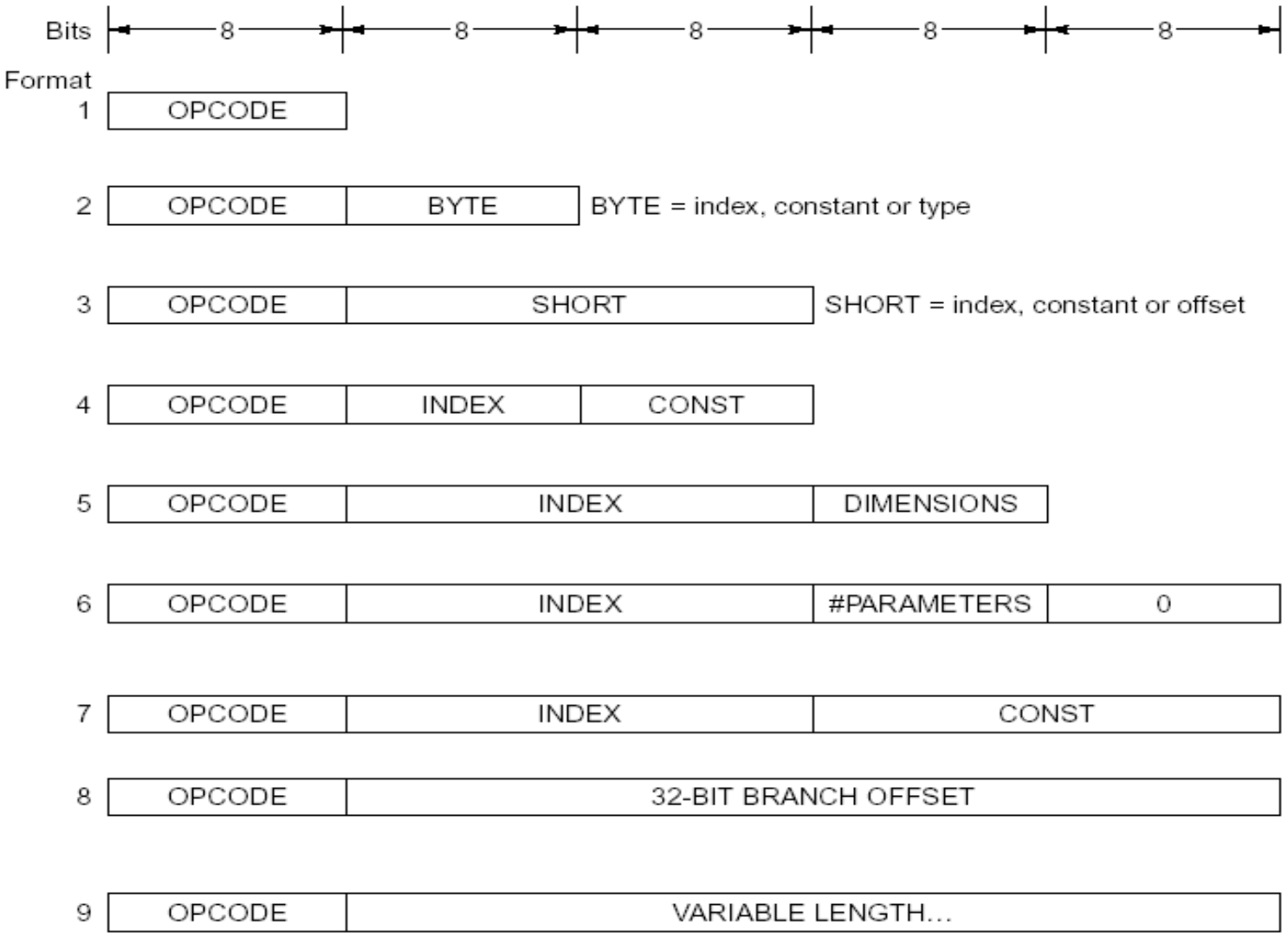
- Flere «kjerner» -> parallellkjøring av program(-tråder)
- Flyttet Northbridge (og GPU) «on die»



LITT MER OM ISA: JAVA BYTEKODE (32 BIT)

For spesielt interesserte

JVM ISA: Format



JVM: 226 instruksjoner

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

ilAND	Boolean AND
ilOR	Boolean OR
ilXOR	Boolean EXCLUSIVE OR
ilSHL	Shift left
ilSHR	Shift right
ilUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable type, x, y = I, L, F, D
CON8/16, D, ATYPE = constant OFFSET16 for branch

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

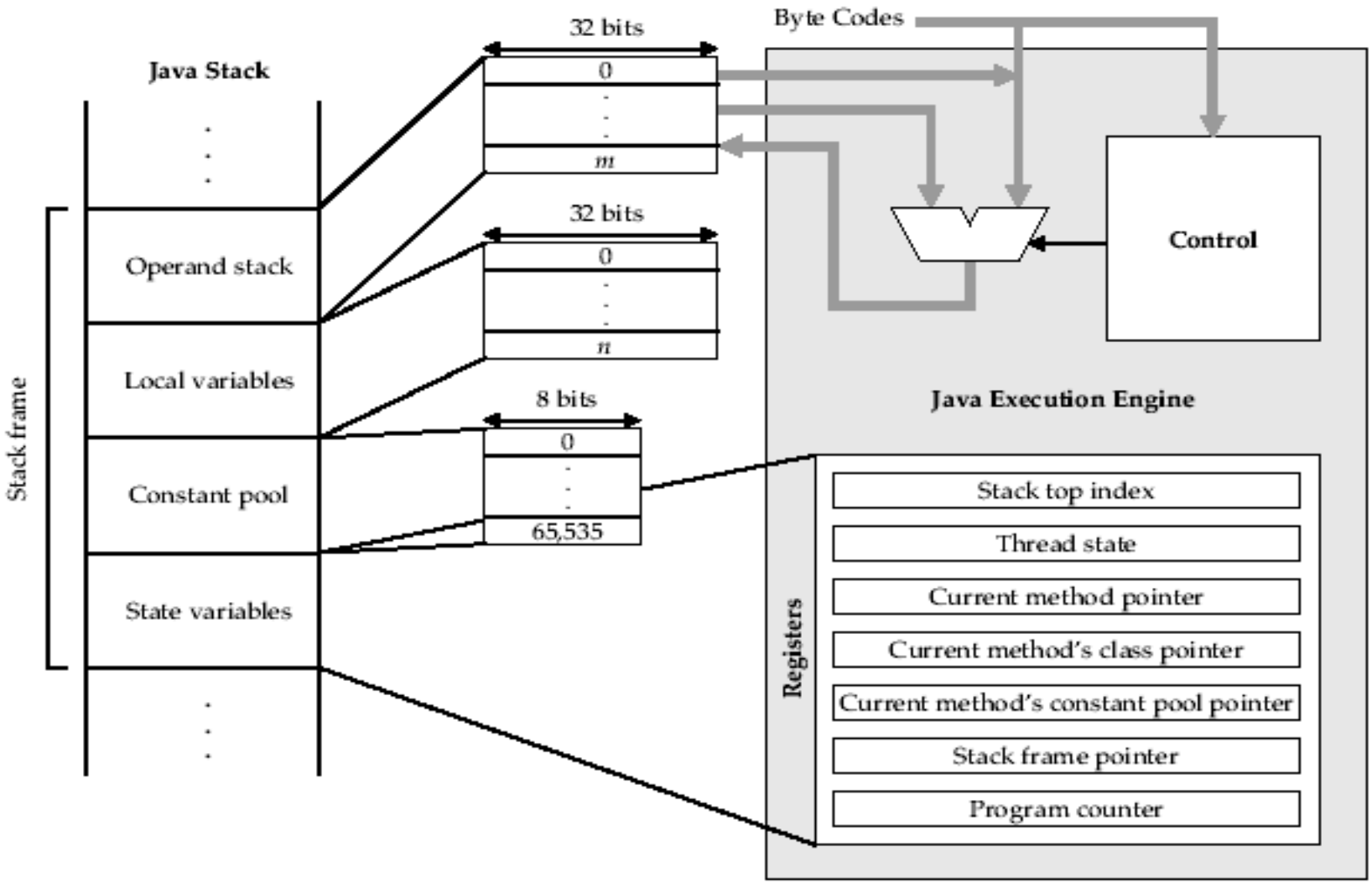
Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Eksempel: JVM (Java RunTime Enviroment)

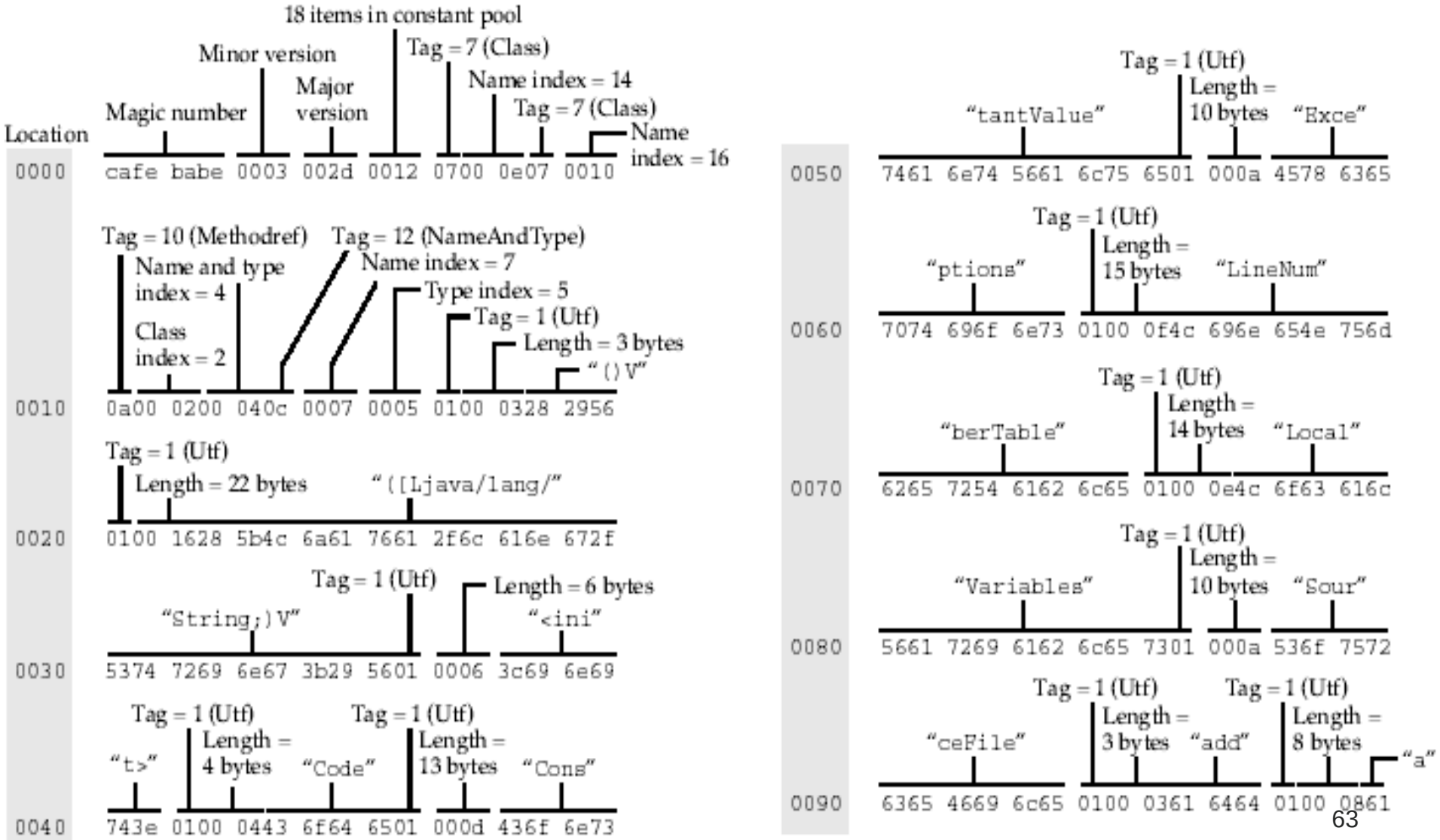


Eksempel: Kode og klasse

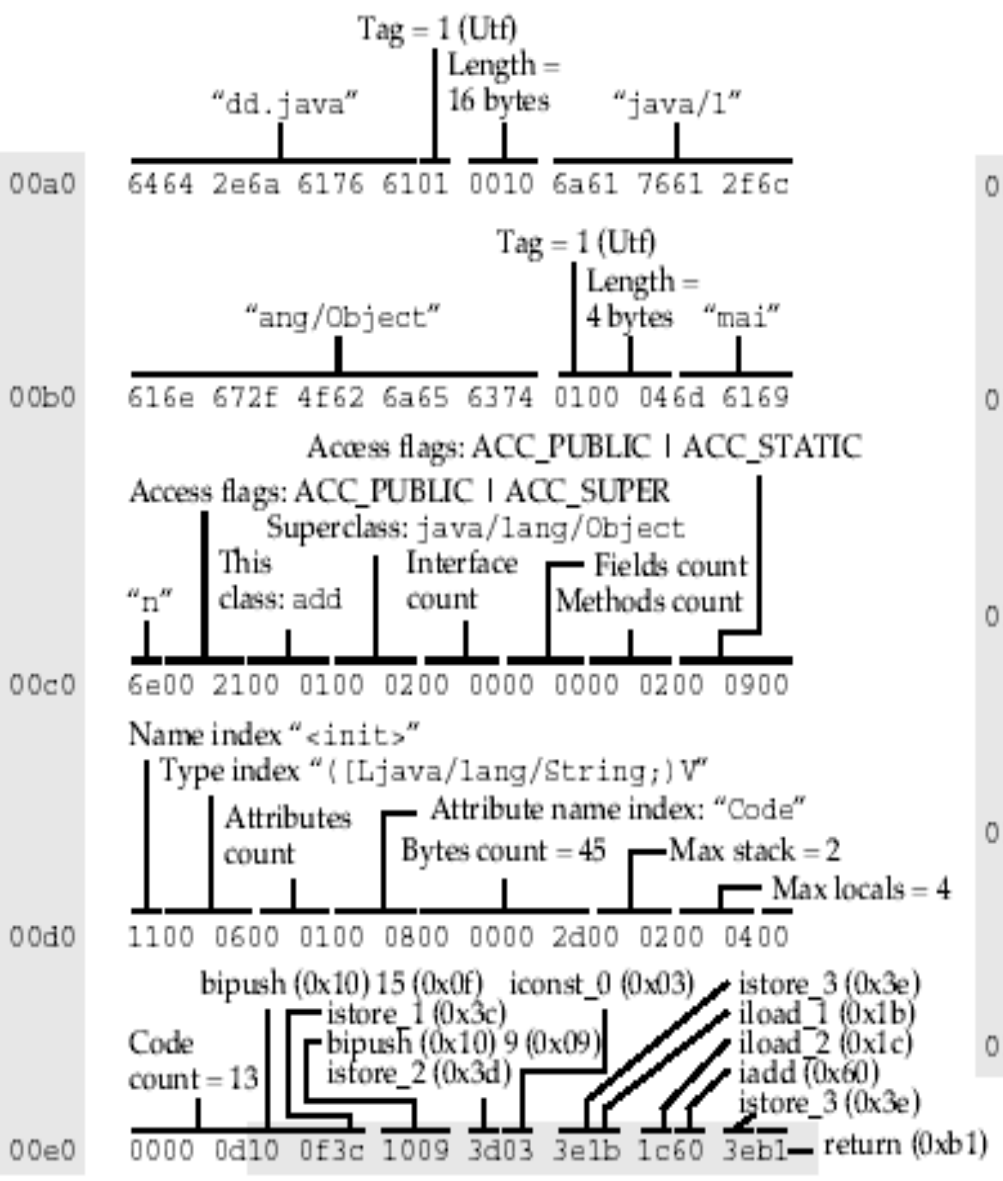
```
public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
        }
    }
```

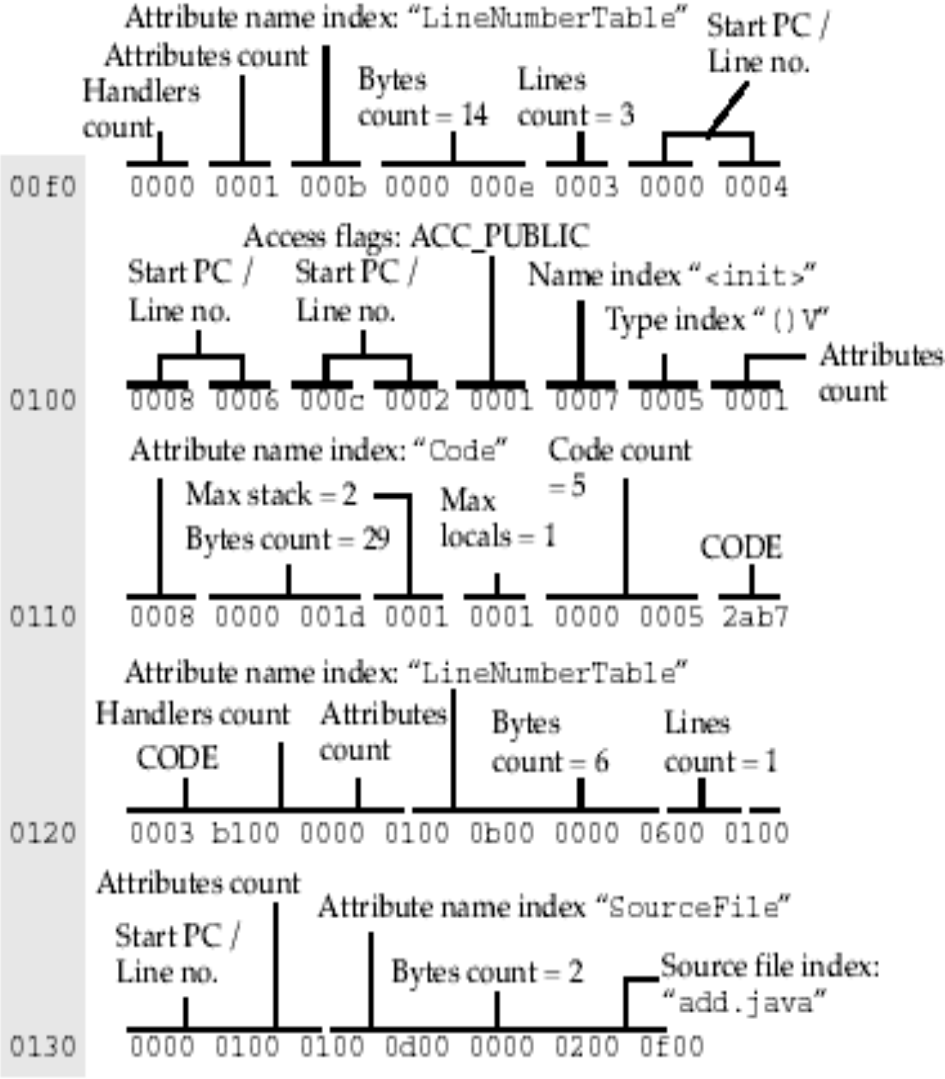
```
0000 cafe babe 0003 002d 0012 0700 0e07 0010   ..............
0010 0a00 0200 040c 0007 0005 0100 0328 2956   ............()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f   ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69   String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73   t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365   tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d   ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c   berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572   Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861   ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c   dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169   ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900   n.!.............
00d0 1100 0600 0100 0800 0000 2d00 0200 0400   ..........-.....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1   .....<..=.>..`>.
00f0 0000 0001 000b 0000 000e 0003 0000 0004   ................
0100 0008 0006 000c 0002 0001 0007 0005 0001   ................
0110 0008 0000 001d 0001 0001 0000 0005 2ab7   ..............*.
0120 0003 b100 0000 0100 0b00 0000 0600 0100   ................
0130 0000 0100 0100 0d00 0000 0200 0f00        ..............
```

Klassefilen (1)



Klassefilen (2)





Byte-koden

Location	Code	Mnemonic	Meaning
0x00e3	0x10	bipush	Push next byte onto stack
0x00e4	0x0f	15	Argument to bipush
0x00e5	0x3c	istore_1	Pop stack to local variable 1
0x00e6	0x10	bipush	Push next byte onto stack
0x00e7	0x09	9	Argument to bipush
0x00e8	0x3d	istore_2	Pop stack to local variable 2
0x00e9	0x03	iconst_0	Push 0 onto stack
0x00ea	0x3e	istore_3	Pop stack to local variable 3
0x00eb	0x1b	iload_1	Push local variable 1 onto stack
0x00ec	0x1c	iload_2	Push local variable 2 onto stack
0x00ed	0x60	iadd	Add top two stack elements
0x00ee	0x3e	istore_3	Pop stack to local variable 3
0x00ef	0xb1	return	Return

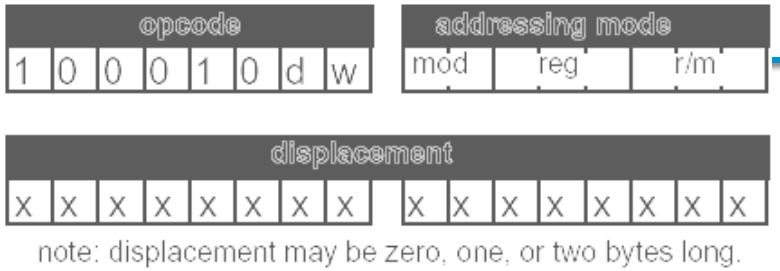
Java bytecode vs x86 instruksjoner

- Java bytecode kan kalles et RISC-type instruksjonssett
- x86 (Intel, så vel som AMD) er CISC

- De to neste foilene viser:
  - De eldste instruksjonene som fremdeles er med (256 av dem, det fins mange, mange flere flere)
  - Eksempel på en IA-32 instruksjon

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD rmb,rb	ADD rmv,rv	ADD rb,rmb	ADD rv,rmv	ADD AL,ib	ADD eAX,iv	PUSH ES	POP ES	OR rmb,rb	OR rmv,rv	OR rb,rmb	OR rv,rmv	OR AL,ib	OR eAX,iv	PUSH CS	386 space
1	ADC rmb,rb	ADC rmv,rv	ADC rb,rmb	ADC rv,rmv	ADC AL,ib	ADC eAX,iv	PUSH SS	POP SS	SBB rmb,rb	SBB rmv,rv	SBB rb,rmb	SBB rv,rmv	SBB AL,ib	SBB eAX,iv	PUSH DS	POP DS
2	AND rmb,rb	AND rmv,rv	AND rb,rmb	AND rv,rmv	AND AL,ib	AND eAX,iv	ES:	DAA	SUB rmb,rb	SUB rmv,rv	SUB rb,rmb	SUB rv,rmv	SUB AL,ib	SUB eAX,iv	CS:	DAS
3	XOR rmb,rb	XOR rmv,rv	XOR rb,rmb	XOR rv,rmv	XOR AL,ib	XOR eAX,iv	SS:	AAA	CMP rmb,rb	CMP rmv,rv	CMP rb,rmb	CMP rv,rmv	CMP AL,ib	CMP eAX,iv	DS:	AAS
4	INC eAX	INC eCX	INC eDX	INC eBX	INC eSP	INC eBP	INC eSI	INC eDI	DEC eAX	DEC eCX	DEC eDX	DEC eBX	DEC eSP	DEC eBP	DEC eSI	DEC eDI
5	PUSH eAX	PUSH eCX	PUSH eDX	PUSH eBX	PUSH eSP	PUSH eBP	PUSH eSI	PUSH eDI	POP eAX	POP eCX	POP eDX	POP eBX	POP eSP	POP eBP	POP eSI	POP eDI
6	PUSHA PUSHAD	POPA POPAD	BOUND rv,m2v	ARPL rm2,r2	FS:	GS:	OpLen	AdLen	PUSH iv	IMUL rv,rmv,iv	PUSH ib	IMUL rv,rmv,ib	INSB	INSD	OUTSB	OUTSD
7	JO ib	JNO ib	JB ib	JAE ib	JE ib	JNE ib	JBE ib	JA ib	JS ib	JNS ib	JP ib	JNP ib	JL ib	JGE ib	JLE ib	JG ib
8	Immed rmb,ib	Immed rmv,iv		Immed rmv,ib	TEST rmb,rb	TEST rmv,rv	XCHG rmb,rb	XCHG rmv,rv	MOV rmb,rb	MOV rmv,rv	MOV rb,rmb	MOV rv,rmv	MOV rm,segr	LEA rv, m	MOV segr,rm	POP rmv
9	NOP	XCHG eAX,eCX	XCHG eAX,eDX	XCHG eAX,eBX	XCHG eAX,eSP	XCHG eAX,eBP	XCHG eAX,eSI	XCHG eAX,eDI	CWDE CBW	CWQ CDQ	CALL FAR s:m	FWAIT	PUSHF	POPF	SAHF	LAHF
A	MOV AL,[iv]	MOV eAX,[iv]	MOV [iv],AL	MOV [iv],eAX	MOVSB	MOVSD	CMPSB	CMPSD	TEST AL,rb	TEST eAX,iv	STOSB	STOSD	LODSB	LODSD	SCASB	SCASD
B	MOV AL,ib	MOV CL,ib	MOV DL,ib	MOV BL,ih	MOV AH,ib	MOV CH,ih	MOV DH,ih	MOV BH,ih	MOV eAX,iv	MOV eCX,iv	MOV eDX,iv	MOV eBX,iv	MOV eSP,iv	MOV eBP,iv	MOV eSI,iv	MOV eDI,iv
C	Shift rmb,ib	Shift rmv,ib	RET i2	RET	LES rm,rmp	LDS rm,rmp	MOV rmb, ib	MOV rmv, iv	ENTER i2, ih	LEAVE	RETF i2	RETF	INT 3	INT ih	INTO	IRET
D	Shift rmb, 1	Shift rmv, 1	Shift rmb,CL	Shift rmv,CL	AAM	AAD		XLATB	87 space	87 space	87 space	87 space	87 space	87 space	87 space	87 space
E	LOOPNE short	LOOPE short	LOOP short	JeCXZ short	IN AL,[ib]	IN eAX,[ib]	OUT [ib],AL	OUT [ib],eAX	CALL iv	JMP iv	JMP FAR s:m	JMP ib	IN AL,[DX]	IN eAX,[DX]	OUT [DX],AL	OUT [DX],eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary rmb	Unary rmv	CLC	STC	CLI	STI	CLD	STD	IncDec rmb	Indir rmv

MOV (386->) – mod-reg-r/m



d og **w** i opkoden definerer lengde og retning på operasjonen

Table 23: REG Bit Encodings

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Table 24: MOD Encoding

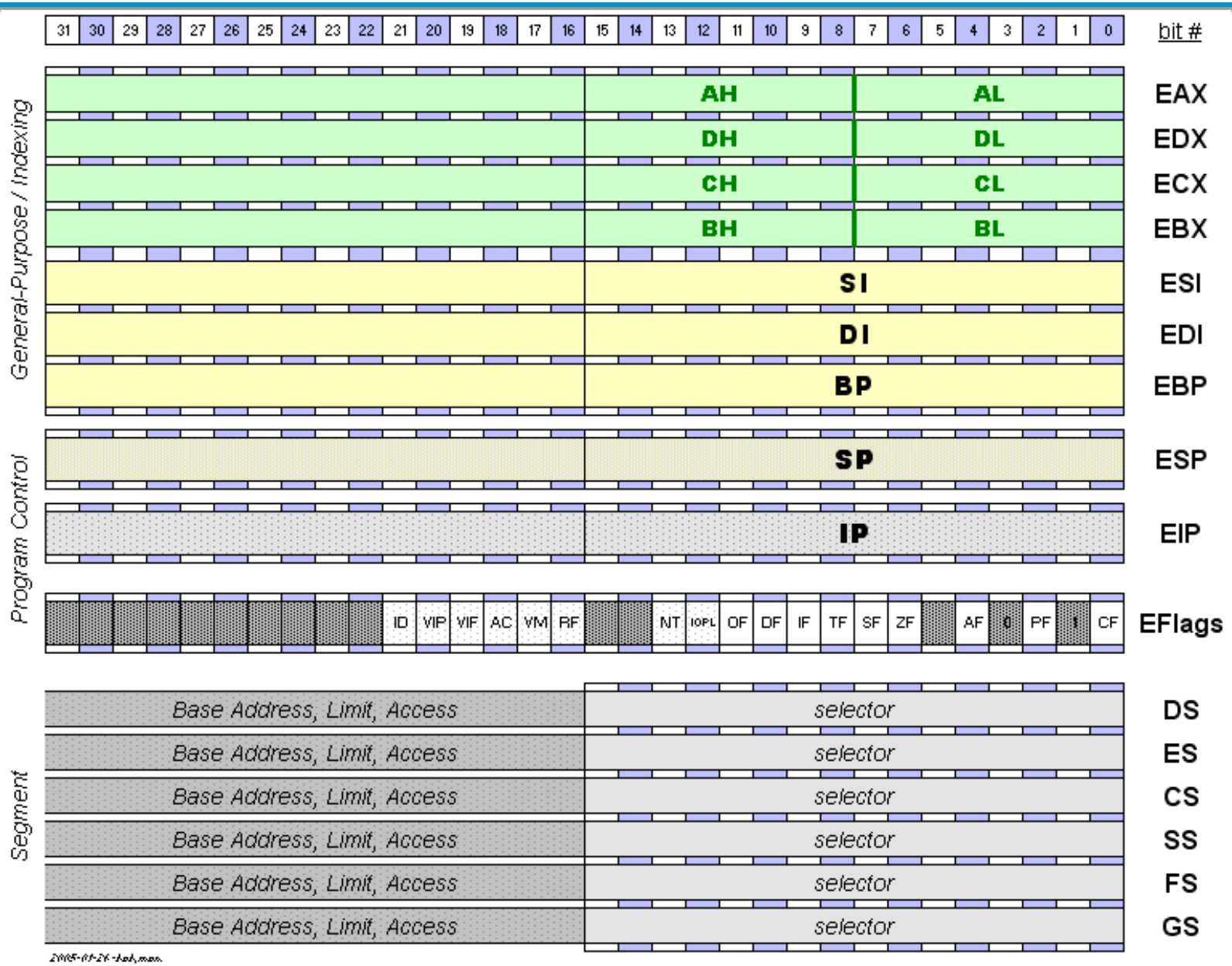
MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

Table 25: R/M Field Encoding

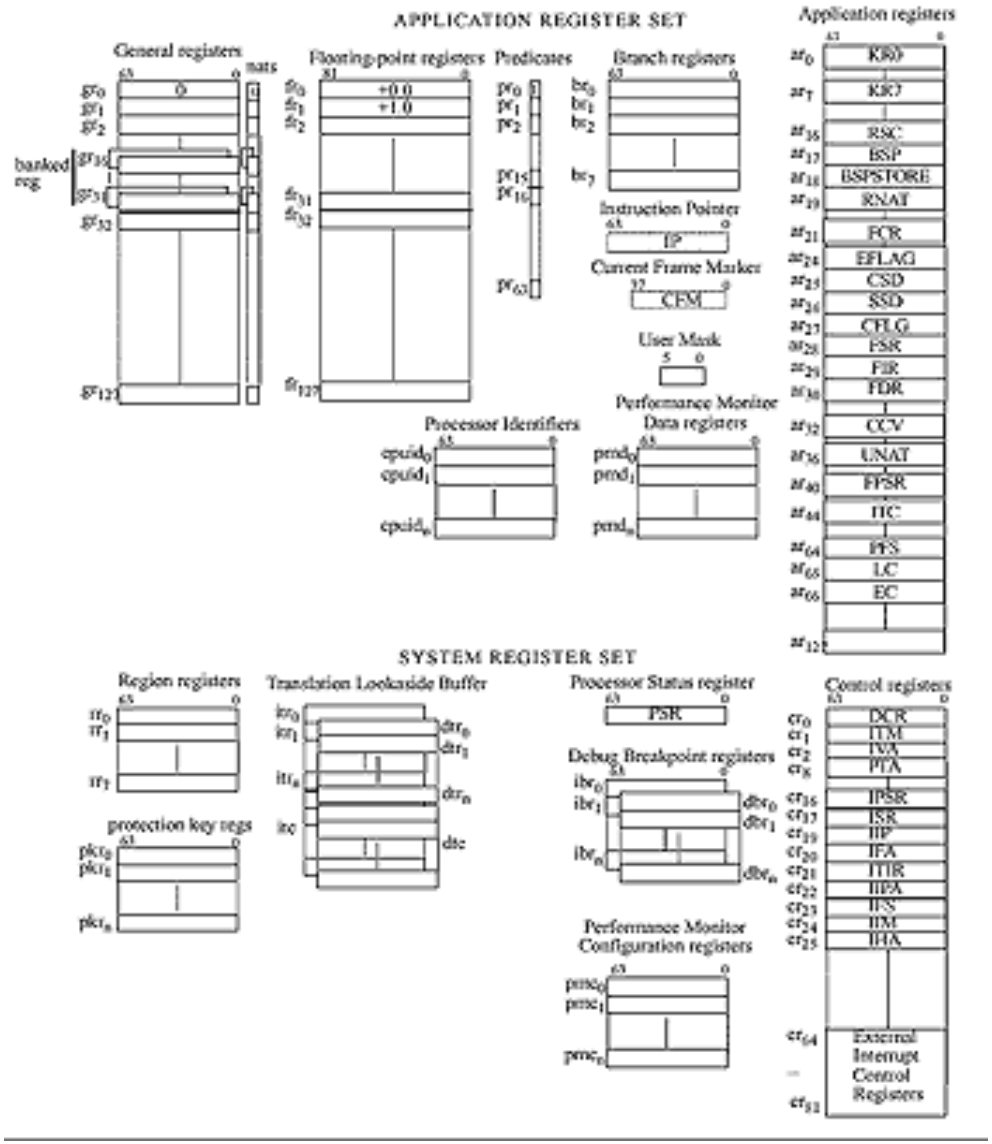
R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

```
[ebx] [ebp]          ;Uses DS by default.
[ebp] [ebx]          ;Uses SS by default.
[ebp*1] [ebx]        ;Uses DS by default.
[ebx] [ebp*1]        ;Uses DS by default.
[ebp] [ebx*1]        ;Uses SS by default.
[ebx*1] [ebp]        ;Uses SS by default.
es:[ebx] [ebp*1]     ;Uses ES.
```

386 registre (IA32) - brukerprogram



X86-64 registre



- Svært mange ulike registre
  - på hver kjerne!
- Ekstrem kompleksitet
- Hvordan forholder vi oss til kompleksiteten?
  - **Abstraherer!**

Abstraksjon

- **Abstraksjon**
  - et begrep eller en ide som ikke er forbundet med noen konkret instans
  - å fjerne det unødvendige i en gitt sammenheng
  - å fokusere på de trekkene som er felles
  - Å «gjemme» kompleksitet
- Eksempel:
  - I et Java-program så kan du gjemme et komplisert program inne i en metode…
    - etter det trenger du bare å forholde deg til metoden
  - Høynivå-språk abstraherer bort all den maskinnære kompleksiteten
    - System.out.print("Hallo World!") involverer i realiteten noen tusen instruksjoner siden er et metode-kall fra java.exe til operativsystemets skrive-til-skjerm-rutiner.

Forskjeller på x86 16, 32 og 64 bit?

- 16 bit arkitektur
  - 16 bits register-størrelse (word)
  - 20 bits adressering (max 1 MiB «RAM»)
    - Segmentert adressering (CS:IP f.eks.)
  - 16 bits ord (typisk 16 bit int f.eks.)
  - Ikke noe skille mellom kjøring av kjerne-kode i OS og vanlige programmer
- 32 bit (386->Pentium)
  - 32 bits register (ord-størrelse)
  - 32 bits adressering (4 GiB Virtuelt minn)
    - Flatt og støtte for virtuelt minne
  - 32 bits ord
  - Flyttall-støtte «on die»
  - Skille mellom kernel- og user-kjøring
- 64 bit (nå)
  - 64 bits registre og flere arbeidsregistre
  - Typisk 48 bits adressering (16 EiB virtuelt minne)
  - Ofte multi-kjerne
  - Enda flere instruksjoner…

AVSLUTNING

Poeng

- CISC
  - COMPLEX Instruction Set Computer
  - Mange forskjellige instruksjoner
  - Instruksjoner har ulik lengde i bit/byte
- RISC
  - REDUCED Instruction Set Computer
  - Bare de instruksjonene som (erfaringsmessig) brukes mye.
  - Gir enklere elektronikk og dermed raskere utførelse.
  - Jf byte-kode på JVM
- Moderne Intel/AMD CPUer
  - ”begge deler”
  - CISC eksternt
  - RISC mikroinstruksjoner på selve prosesoren

Flere poeng

- Prosessoren kjører instruksjonene i den rekkefølgen de ligger i minnet («**sekvensielt**»).
- På **ulike datatyper** benyttes helt **forskjellige instruksjoner**
  - 7 + 5:
    ```
    mov AX,7
    add AX,5
    ```
    resulterer i at det ligger 12 i AX-registeret
  - 7.0 + 5.0:
    ```
    FLD «minneadressen der 7.0 ligger»
    FLD 0012AC45
    FADD ST, ST(1)
    FSTP «adressen svaret skal legges»
    ```
  - 'Hei' + 'Hå!'
    mengdevis av instruksjoner som flytter begge strengene inn i felles minneområde og legge til slutt-markøren (typisk ASCII 0x00).

Flere poeng

- Hva er en variabel?
  - På instruksjonsnivå: **adressen** til et sted minnet (eller et register)
- Hvorfor har variabler «scope»
  - Å kalle en metode medfører at man begynner å kjøre instruksjoner et helt annet sted i minnet
  - Når man hopper tilbake til der metoden ble kalt forsvinner alt annet enn det som metoden returner (typisk en adresse til en verdi…)

Hva skal vi kunne?

- Boolsk algebra
  - NOT, AND, OR, XOR
- Hva en *instruksjon* er.
  - hvordan bygd opp
  - hvilke typer
- Hva et *register* er.
  - hvilke **typer** registre som finnes på CPU
  - Forskjellen på å adresse- og svar-register
- Von Neumann modellen (igjen)
  - mindre «black box»
- Forskjellen på CISC og RISC

Hva skal vi kunne?

- Litt om x86 og alternativer
- Rollen til taktgiver
- Fetch-Execute-syklusen
- Pipelining og superskalaritet
- Ulike anvendelser av og typer CPUer/mikrokontrollere