# Finn Schiermer Andersen

Ekstern lektor, DIKU
Leder udviklingen af `realm-core`, https://realm.io/
(Git repo: https://github.com/realm/realm-core)

## Tidligere:

IOInteractive: Hitman Absolution, Glacier2 gaming engine
Oticon: digital platform (chips til høreapparater)
Thrane: Satelit kommunikation
Mips Technologies: Microprocessor design

# Oversigt over forelæsningerne i maskinarkitektur

1. En essentiel maskine bygget af nogle passende byggeklodser.
Så simpelt som muligt - men ikke simplere. A2.

2. Et deep dive i hvordan klodserne bygges.
Masser af detaljer der giver baggrund

3. Pipelining - hvorfor og hvordan?
Performance! Mere performance! Hvor langt kan man gå? Hvordan?
Mere realisme

4. Avanceret mikroarkitektur.
Parallel udførsel af sekventiel kode. Hvordan?
"The bureaucracy is expanding to meet the need of the expanding bureaucracy"

5. Multitrådning. Multicore. Opsamling.

# Abstraktionsniveauer

1. Gode programmeringssprog: Erlang, OCaml osv

2. Maskinnære programmeringssprog: C

3. Assembler / Symbolsk Maskinsprog: x86, ARM, MIPS

4. Arkitektur (ISA): Maskinsprog - ordrer indkodet som tal

5. Mikroarkitektur: ting som lager, registre, regneenheder, afkodere
og hvordan de forbindes så det bliver en maskine

6. Standard celler: Simple funktioner af få bit (1-4) med et eller to resultater. Lagring af data (flip-flops)

7. Transistorer

8. Fysik. Eller noget der ligner

# Den Simpleste Maskine

Næsten

x86 bliver aldrig helt simpel. Men vi vil forsøge.

# Vi bruger et lille udsnit af x86 assembler

1. Aritmetik - kun mellem registre
2. 2-komplement aritmetik: addq, subq, cmpq
3. Bitvis logik: andq, xorq
4. Kontrol
5. betinget hop: Jcc (jmp,je,jne,jle,jl,jge,jg)
6. funktionskald: call, ret
7. stop verden! jeg vil af: hlt
8. Data flytning (egentlig er det kopiering, men...)
9. konstant til register: movq $imm, %reg
10. register til register: movq %reg, %reg
11. betinget kopiering: cmovcc %reg,%reg
12. register til lager: movq %reg, displacement(%reg)
13. lager til register: movq displacement(%reg), %reg
14. stak operationer: pushq %req, popq %reg
15. fordi vi kan: nop

# Med en simplere indkodning

| | |
|---|---|
| halt | 0x00 |
| nop | 0x10 |
| movq %a,%b | 0x20 ab |
| cmovcc %a,%b | 0x2c ab |
| movq $i,%b | 0x30 0b ii ii ii ii |
| movq %a,i(%b) | 0x40 ab ii ii ii ii |
| movq i(%b),%a | 0x50 ab ii ii ii ii |
| OPq %a,%b | 0x6q ab |
| jcc t | 0x7c tt tt tt tt |
| call t | 0x80 tt tt tt tt |
| ret | 0x90 |
| pushq %a | 0xA0 a0 |
| popq %a | 0xB0 a0 |

Bemærk 'c' og 'q'. 'c' angiver betingelse for betinget hop eller betinget move 'q' angiver hvilken aritmetisk operation der skal udføres

# Indkoding af 'c' og 'q'

c: (betingelse)

0 always (jmp, movq)
1 less or equal (jle, cmovle)
2 less (jl, cmovl)
3 equal (je, cmove)
4 not equal (jne, cmovne)
5 greater or equal (jge, cmovge)
6 greater (jg, cmovg)

q: aritmetisk operation

0 addq
1 subq
2 andq
3 xorq
4 cmpq

# Nyttige programmer i en svær tid

Fra https://github.com/kirkedal/compSys-e2017-sim

* En assembler som kan indkode vore programmer

* Angiv et assemblerprogram som argument - viser indkodningen

* Angiv endvidere et ekstra filnavn, og det indkodede program placeres der.

* En simulator som kan udføre de indkodede programmer

* Angiv et indkodet program som argument - viser sluttilstand på registre og lager

* Angiv endvidere et ekstra filnavn, og side-effekter fra kørslen af programmet placeres der. Vi vil senere kalde denne fil for en sporings-fil.

* Et program der kan udskrive listen af side-effekter i lidt mere menneskevenlig form

Sku' vi kigge på det?

# A2 - Jeg har et tilbud til dig!

* Vi giver jer en simulator der ikke er helt færdig

* En simulator er et program som lader som om det er en maskine. Det vil sige: det kan udføre andre programmer skrevet til den maskine der simuleres.

* Senere giver I os en simulator der *er* færdig. Den kan udføre den begrænsede mængde af x86 ordrer jeg lige har beskrevet

* Der er krav til hvordan i skriver jeres simulator. De krav afspejler hvordan hardware essentielt virker. Så lad os uddybe det.

# Hardware - helt essentielt

Der er grundlæggende to slags byggeklodser:

* Funktionelle byggeklodser. De beregner hele tiden et resultat som funktion af input. Eksempler:
    * Adder (lægger tal sammen)
    * Aritmetisk-logisk enhed (kan også lave bitvis and/or, evt skifte)
    * Multiplexor (vælger et af flere muligheder)
    * Afkoder (sætter styresignaler ud fra input)
    * Kombinationer af and, or, not

* Tilstandselementer. Denne slags byggeklodser kan "huske" data. De opdateres på fastlagte tidspunkter, synkroniseret af en puls, på nudansk kaldet en "clock." Eksempler:
    * Register
    * Register-blok (eller register-fil)
    * Lager-blok (lidt som register-blok, men større og langsommere)

De kan **alle** bygges af nand eller nor elementer. Mere herom næste uge.

# Hardware - helt essentielt (2)

    * Byggeklodser kan også være sammensatte. I så fald har de en eller flere "indre" forbindelser eller funktionalitet af hver type. Et typisk eksempel er en lagerblok. Den består af tilstandselementer, men har også en eller flere "læse-porte" som tager en adresse som input og giver indholdet af lagercellen på adressen som output.

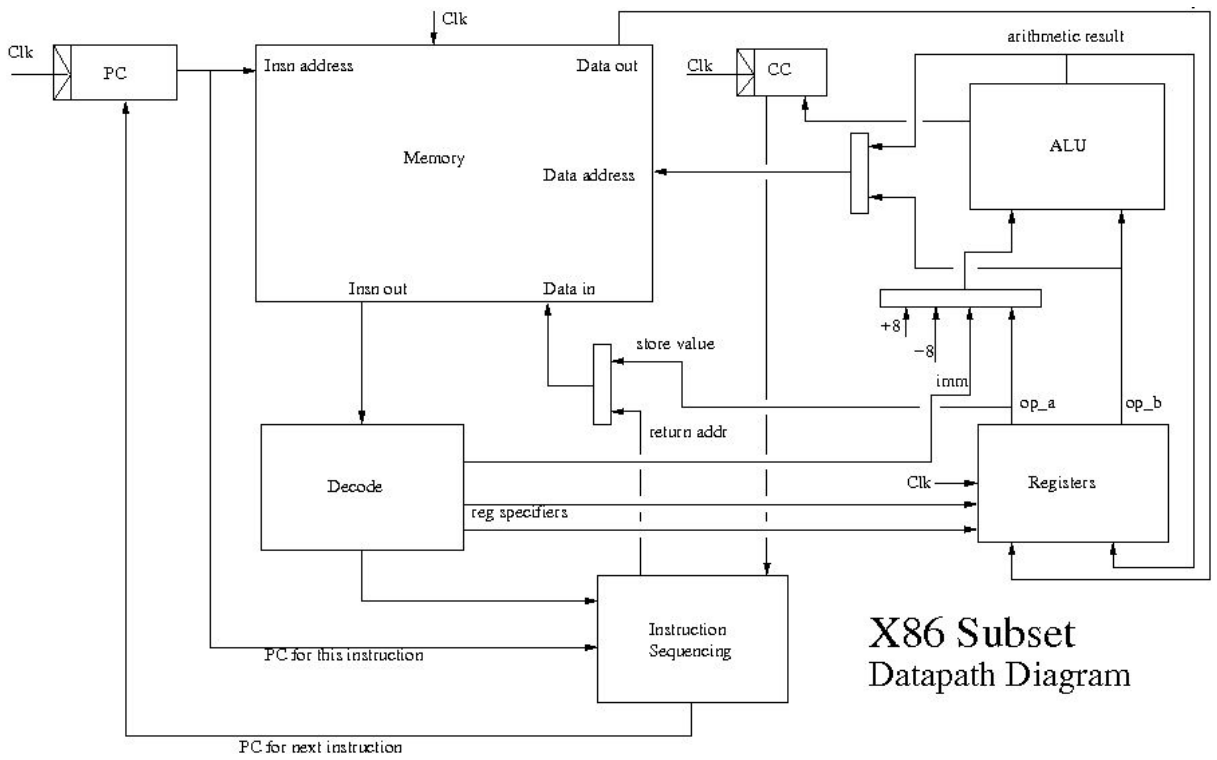
Byggeklodserne kan forbindes, så output fra en byggeklods flyder til input på en anden byggeklods.

Hvis du forbinder funktionelle byggeklodser (altså dem der ikke kan lagre resultater), så de er cirkulært afhængige, så bliver resultatet udefineret.

    * Hvorfor det?

Der er **altid** cirklære afhængigheder i en mikroarkitektur, men **alle** cirkulære afhængigheder skal splittes ved brug af tilstandselementer.

# Eksempel på en mikroarkitektur



# Forklaringer/Noter til mikroarkitektur

* Alle byggeklodser med et "Clk" signal er/indeholder tilstandselementer.

* F.eks. er "PC" et register der indeholder programtælleren

* Byggeklodserne "Memory" og "Registers" er primært tilstandselementer, men de kan *læses* på ren funktionel vis.

* De flade kasser uden titel kaldes "multiplexere" eller "muxe". De vælger mellem forskellige inputs og sender det valgte videre.

* Byggeklodsen "Instruction Sequencing" er ansvarlig for at beregne adressen på den næste instruktion der skal udføres.

* "Decode" er ansvarlig for at splitte data fra "Memory" op i forskellige dele som matcher indkodningen for instruktioner i vores maskine

* "ALU" er ansvarlig for at lave aritmetiske beregninger. Blandt. andet.

Bemærk at mange kontrol-signaler ikke er vist i diagrammet.

# Eksempel: flow for NOP-instruktionen

1. Som resultat af en puls på "Clk" bliver PC opdateret. Den nye værdi drives fra PC til Memory og Instruction Sequencing

2. Memory responderer ved at drive instruktionen på den udpegede adresse til Decode.

3. Decode genererer styresignaler til resten af datavejen. I det her tilfælde er den hentede instruktion en Nop, så noget af det, Decode skal gøre er at sikre at registre og lager ikke opdateres. Decode skal også bestemme størrelsen af instruktionen, i det her tilfælde 1 byte. Det skal sendes til Instruction Sequencing.

4. Instruction Sequencing skal beregne den næste værdi til PC. Det gøres ved at lægge instruktionens størrelse til den gamle værdi. Resultatet sendes tilbage til PC.

5. Først når en ny puls ankommer på "Clk" vil PC registeret blive opdateret og udførelse af den næste instruktion kan starte. Indtil da vil kredsløbet falde til ro.

# Eksempel: flow for ADDQ %ra,%rb

1. Vi starter som for NOP, men Decode opfører sig anderledes for ADDQ. For det første skal skrivning til registrene slås til. For det andet skal numre på register a og b fiskes ud af instruktionen og sendes til "Register". Det er også nødvendigt at styre ALU'en så den udfører en addition.

2. Register udlæser de to operander fra de angivne registre og føder dem til ALU’en.

3. ALU’en udfører addition og beregner nyt indhold til CCR (condition code register)

4. Resultatet fra ALU’en føres tilbage til register-filen

5. Kredsløbet falder til ro

6. En ny puls på Clk når både til PC (og fører til opdatering til næste instruktion) og Register og fører til opdatering af destinations registeret.

# Opsamling

* Alle de funktionelle byggeklodser er altid aktive. Ændret input fører til beregning af nyt output.

* Byggeklodser der ikke indgår i udførelsen af en given instruktion er alligevel aktive. Man skal blot sikre at det ikke fører til opdatering af tilstandselementer med forkerte resultater.

* Hver clock-cyklus starter med en puls som opdaterer registre og lager.

* Derefter "løber" beregningen gennem de funktionelle byggeklodser, indtil alle signaler er stabile. Så er kredsløbet faldet til ro.

* Derpå kan en ny clock-cyklus starte.

Hvad mon bestemmer clock-frekvensen for sådan en maskine?

# Modellering i C

Vi har oversat egenskaberne ved byggeklodserne og hvordan de kombineres til *formkrav* til programmer skrevet i C. Det ligner ikke normalt C.

Vi bruger kun to typer i vores program: en til kontrol-signaler, en til data: * Vi repræsenterer et kontrol signal med en "bool" * Vi repræsenterer øvrigt data med en "val"

En maskin cyklus udtrykkes ved et løkke gennemløb. Tilstandselementer erklæres udenfor løkken. Alle de funktionelle byggeklodser udfører deres arbejde først i hvert gennemløb. Alle tilstandselementer opdateres i slutningen af hvert gennemløb.

```
val PC;
while (..) {
        // funktionelle byggeklodser:
        val PC_out = PC;
        val next_PC = add(PC_out, from_int(1));
        // clock-puls... opdatering af tilstandselementer:
        PC = next_PC;
}
```

# Udleverede byggeklodser

Vi udleverer en bunke byggeklodser til brug i opgaven.

* wires.h/c - funktioner der svarer til at "trække ledninger"

* arithmetic.h/c - grundlæggende aritmetik og bitvis logik

* alu.h/c - aritmetisk/logisk enhed skræddersyet til vores maskine

* memories.h/c - lagerblokke: registre, program- og data-lager

Lad os se nogle eksempler

# Forbindelser/ledninger

```
// simple conversion
val from_int(uint64_t);

// pick a set of bits from a value
val pick_bits(int lsb, int sz, val);

// pick a single bit from a value
bool pick_one(int position, val);

// sign extend by copying a sign bit to all higher positions
val sign_extend(int sign_position, val value);
```

Og flere, men ovenstående er de mest betydende

# Logiske operationer

For kontrol signaler, repræsenteret ved bool's bruger man bare de indbyggede logiske operatorer.

For data, repræsenteret ved typen val finds der følgende funktioner

```
// mask out a value if control is false
val use_if(bool control, val value);

// bitwise and, or, xor and negate for bitvectors
val and(val a, val b);
val or(val a, val b);
val xor(val a, val b);
val neg(int num_bits, val);

// reduce a bit vector to a bool by and'ing or or'ing all elements
bool reduce_and(int num_bits, val);
bool reduce_or(val);

// 64 bit addition
val add(val a, val b);
```

Vi har også en færdigbagt ALU der tilfældigvis passer til vores maskine

```
typedef struct {
	bool of;
	bool zf;
	bool sf;
} conditions;

bool eval_condition(conditions cc, val op);

typedef struct {
	val result;
	conditions cc;
} alu_execute_result;

alu_execute_result alu_execute(val op, val op_a, val op_b);
```

Lad os kigge på den udleverede simulator for A2

for et mere detaljeret eksempel

# Fejlfinding

Som nævnt tidligere kan værktøjet 'sim' (også kaldet reference-simulatoren) producere en sporings-fil - en fil med de sideeffekter et program har mens det udføres.

Den udleverede simulator til a2 (den I skal færdiggøre) kan læse sådan en sporings-fil og sammenholde den med hvad der sker under simulationen.

Hvis der detekteres en afvigelse fra "sporet", kaldes en funktion der hedder error() med en fejlmeddelelse. Meddelelsen skrives ud og programmet terminerer.

Men man kan køre simulatoren i gdb og sætte et breakpoint på error(). Derefter er det ligetil at inspicere variable i programmet

## Skal vi prøve?

# Hvad nu?

Hardware kan på det her niveau opfattes som en række byggeklodser forbundet til hinanden via kanaler. Kommunikationen i en kanal er envejs og altid i samme retning.

A2 går ud på at demonstrere forståelse for emnet ved at lave en model af en maskine, en simulator, i en begrænset udgave af C. Begrænsningerne er udtrykt ved formkrav der afspejler de grundlæggende egenskaber ved hardware.

* Leg med de udleverede værktøjer. Skriv små x86 programmer.
* Gå til øvelser - de er relevante for A2
* Få et overblik over det der er udleveret (læs header filerne)
* Undersøg den halvfærdige simulator. Forstå hvordan main.c virker.

## Husk at stille spørgsmål

# Spørgsmål og Svar

Vi lader billedet stå et øjeblik: