

TSEA28 Datorteknik Y (och U)

Föreläsning 8

Kent Palmkvist, ISY



Dagens föreläsning

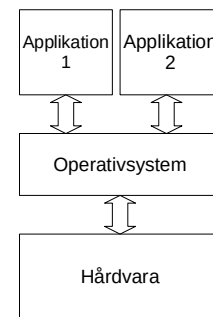
- Minneshantering, virtuellt minne
- Ett större exempel av signalbehandlande system
 - Optimering av kod
 - Nya instruktioner
- Jämförelse med andra processorfamiljer
 - Instruktionsuppbyggnad, maskinkod
- Prestandajämförelser

Praktiska kommentarer

- Labutrustning i MUX1 tas bort under tentaperioden
 - Labbet används i andra kurser
 - Distansutrustning kommer finnas kvar i MUX2
- Extra redovisningstillfälle för Lab 1-3 under omtentaperiod i Juni
 - Info om datum och tid skickas ut via mail

Funktion hos operativsystem

- Operativsystem hanterar hårdvara och gömmer detaljer från användaren
 - Ger illusionen av att flera saker sker samtidigt
 - Delar på resurser så att alla användare får tillgång till dem
 - Applikationen ska inte behöva veta alla detaljer om hårdvaran
- Säkerhet
 - Användarprogram får inte krascha datorn (stänga av funktioner, störa andra program)
- Flera program
 - Förra föreläsningen visade enkelt sätt
 - Avbrott för att byta till nästa program
 - Krävde olika adresser till varje program (inte bra)



Add dela minne

- Grundläggande resurs: Minne
- Programmets placering i minnet styrs av operativsystemet
 - Laddar in program från sekundärminne (hårddisk, SSD, eller liknande)
- Varje program vill ha tillgång till minne utan att bry sig om andra programs behov
 - Operativsystemet ska hantera tillgängligt minne
 - Programmet begär viss mängd minne från operativsystemet och får en adress tillbaks
 - Programmet lämnar tillbaks minne till operativsystemet när uppgiften är klar

Flertal problem behöver lösas: Säkerhet

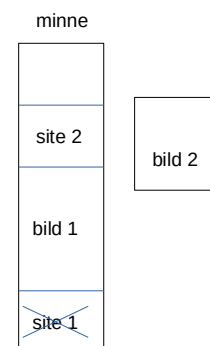
- Säkerhet
 - Olika program ska inte se varandras (eller operativsystemets) minnesinnehåll
- Placering av program i minnet
 - Alla program behöver se samma startadress etc.
- Tillgång till stora mängder kontinuerligt adressutrymme
 - Program vill kunna allokera stora datastrukturer
- Behövs hårdvarulösningar till dessa problem

Flertal problem behöver lösas: Säkerhet

- Säkerhet
 - Olika program ska inte se varandras (eller operativsystemets) minnesinnehåll
- Exempel dålig säkerhet
 - Du kör din internetbank
 - Ett annat program som startats av någon annan (virus/malware). Detta program tittar i minnet på din webbläsare
 - Stor risk att nycklar och passord hittas, t ex tangentbordbuffertar, lokala variabler etc.
- Måste kunna skydda minnesareor från olika användare
 - Varje användares minne ska bara vara tillgängligt för denna användare och för operativsystemet

Flertal problem behöver lösas: Fragmentering

- Även om datorn har tillräckligt med minne jämfört med hur mycket minne varje program behöver så räcker inte det
- Exempel (webbläsare och bildvisare kör samtidigt)
 - Webbläsare allokerar minne för site 1 (20% av totalt)
 - Bildvisare allokerar minne för bild 1 (40% av totalt)
 - Webbläsare allokerar minne för site 2 (20% av totalt)
 - Webbläsare stänger site 1 (totalt 60 % allokerat)
- Bildvisare allokerar minne för bild 2 (30%) <- går inte
 - 40% finns tillgängligt, men delat på två ställen
- Minnesfragmentering!

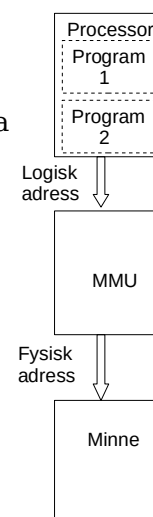


Minnesskydd

- Lägg till funktion (hårdvara) som spärrar tillgång till vissa adresser för andra program
 - Minska risken att ett urspårat program skriver över någon annans data
 - Ta bort möjlighet för fel program läsa känslig information
- Lägg till skrivskydd på vissa minnesytor
 - Kan dela kod utan risk för påverkan
 - Vanligt för standardbibliotek (subrutiner) i datorer
- Val av vilka områden som är tillgängliga styrs av operativsystemet

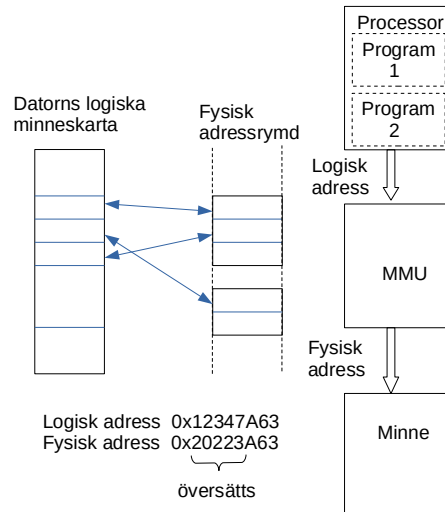
Lösning på säkerhet, minnesskydd och fragmenteringsproblemet

- Memory Management Unit (MMU)
 - Håller ordning på vilket program har tillgång till vilken minneyta
 - Varje program ser en logisk beskrivning av minnesrymden
- Skydda vissa minnesytor mot skrivning av program
- Kan döpa om adresser
 - Flera program kan se det som att dom skriver på adress 0x1000
 - Inga problem samla ihop minnesdelar till stor sammanhållen adressrymd
- Operativsystemet ställer in översättningstabellerna i MMU när program byts



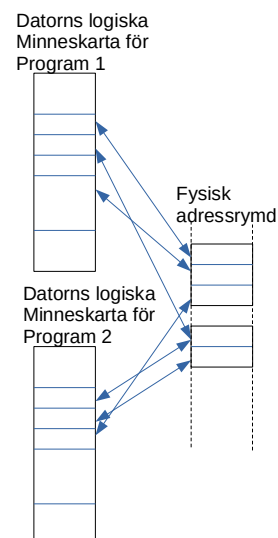
MMU funktion

- Se fysiskt minne som byggblock
 - T ex 4 KByte stora områden (12 addressbitar)
- MMU översätter logisk adress till fysisk adress
 - Högsta adressbitarna väljer vilket block (20 bitar)
 - Lägre bitar används direkt till minnet
 - Olika översättning för varje program



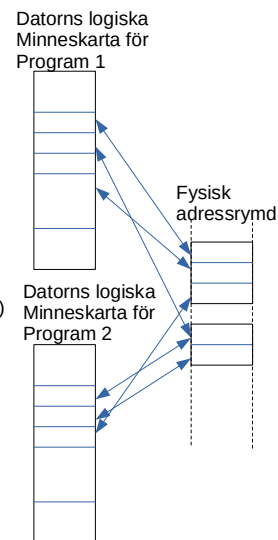
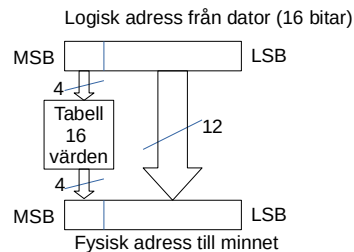
Fördelar med MMU

- Minnesmängd och logiskt minnesutrymme behöver inte matcha varandra
- Slå ihop minnesfragment till sammanhängande minnesytor
- Minnesskydd, tillåt bara en användare komma åt speciella fysiska minnesområden
- Dela minne mellan användare
 - subrutinbibliotek, datastrukturer
- Slipper ändra adresser i programmet (alla program tillåts börja på samma logiska adress)



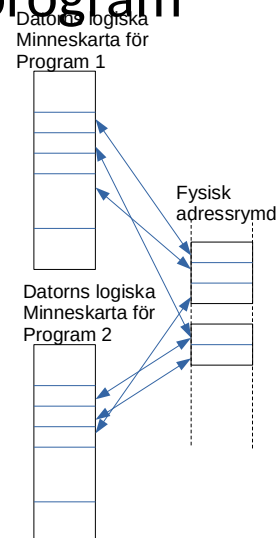
Litet exempel på MMU-funktion

- Antag 16-bit adress (65536 adresser), 16 olika minnesblock
 - Varje block är 4 Kbyte ($65536 / 16$)
 - Översätt bara 4 MSB bitar av 16-bitars adress ($16 = 2^4$)
- En tabell för varje program
 - Operativsystem byter tabell när nästa program får köra



Litet exempel på MMU-funktion, 2 program

- Fysisk position för data och program
 - Program1 adress 0x2000, Program2 adress 0x3000
 - Subrutiner adress 0x4000
 - Program1 data area adress 0x5000, Program2 data area adress 0x8000
- Logisk adress förväntad av programmet (samma för båda)
 - Körs på adress 0x1000, subrutiner på adress 0x2000, data på adress 0x6000
- Två olika minneskarter, en för varje program



Litet exempel på MMU-funktion, forts.

- 16 positioner i översättningstabellen (4 bit)
 - Programmet skrivs för den logiska minneskartan
 - Exempel program1 (logisk start 0x1000)
 - Fysisk start 0x2000

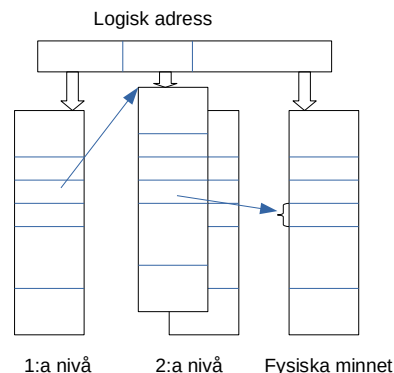

```
mov r0,#0x6330 ; data från fysisk 0x5330
ldr r1,[r0]
bl 0x2010 ; subrutinanrop till 0x4010
b 0x1003 ; hopp till 0x2003
```
 - Program 2: (fysisk start 0x3000)
- Välj rätt tabell beroende på vilket program ska köra
 - Väljs av operativsystemet

Prog1. MMU	
Adr	Data
0	
1	2
2	4
...	
6	5
...	
F	

Prog2. MMU	
Adr	Data
0	
1	3
2	4
...	
6	8
...	
F	

Översättningstabeller för större processorer

- Antag 32-bitars adress, 4KByte block (12 bitar)
 - 20 bitars översättning => väldigt stor tabell
 - 1 miljon entries, dyr och långsam
 - Inte alla program behöver alla adresser
- Praktiskt system delar upp tabellen i två hierarkiska nivåer
 - 10 bitars topnivå
 - 10 bitars mindre tabell
- Också vanligt ha större minnesblock
 - Large och Huge pages (upp till 1GB block)



Expanderad MMU funktion

- Ge avbrott till operativsystemet om referens till tabellrad utan innehåll (page fault)
 - Starta med en tabell för ett program med bara programminnet allokerat
 - Allokera av befintligt minne för att ge program minne bara när dom behöver det
- Kombinera tabelluppslagning med information om det är skrivning
 - Kan få avbrott om försöker skriva till skrivskyddat område
 - Om skrivning görs till delat minnesområde (delat mellan två program) skulle en kopia kunna skapas som programmet skriver till istället

Ge program mer minne än det finns (swap)

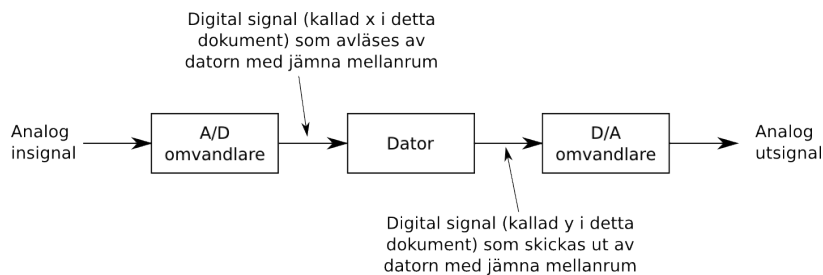
- Använd sekundärminne för att lagra minnesinnehåll som inte används just nu
 - Andra program som inte körs
- Om ett annat program ska fortsätta som placerats på sekundärminnet: flytta tillbaks det till primärminnet
 - Kan kräva mycket läsning/skrivning vid programbyte
- Om ett program använder mer minne än som finns
 - Flytta delar av minnet som programmet inte använt på ett tag till sekundärminne
 - När en adress ska användas som är markerad som utflyttad till sekundärminnet så hämtas den
 - Kan ge väldigt mycket trafik till sekundärminnet

Ett större programexempel

- Hur omskrivningar av kod kan öka prestanda

Ett större programexempel, signalbehandling

- Signalbehandlingsexempel
 - Filtrering av en signal
 - Se länk från föreläsningstabellen för mer information
- Generell struktur



Ett större exempel, forts.

- Antag vi vill bygga ett filter som implementerar

$$y[n] = \sum_{i=0}^N x[n-i] b_i$$

- Antag $b = [-0.0448 \ 0.1659 \ 0.8105 \ 0.1659 \ -0.0448]$
- Lagra koefficienterna i b i en minnesarea coefficients (multiplicerade med 2^{15} för att få heltal)
- Data är 16-bitars värden (2 byte var)
- Antag 100 sampel som ska filtreras per gång är lagrade i en minnesarea 0x20001000 - 0x200013EC
 - 4 extra insampel för att skapa 100 utsampel
- Antag resultatet ska lagras i minnesarea 0x20002000 - 0x200023E8
- Totalt 0x10000 (= 65536) block ska filtreras

Ett större exempel, kodstruktur

- Initiera adresspekare (R5, R6), sampelräknare (R4) samt blockräknare (R3)
- Loop över alla sampel
 - Sätt koefficientpekare R7 till start av koefficienter
 - Nollställ mellanresultat (R2)
 - Loop över alla koefficienter
 - Multiplicera sampel med koefficient => R0
 - Addera till mellanresultat (R2)
 - Hoppa ur loop om adresspekare = coefficient_end
 - Dividera med 2^{15} (för att få multiplikation med 0.0448 etc), spara resultat i ut
 - Räkna ned sampelräknare (R4), ta nästa sampel om inte slut
- Räkna ned block (R3), om alla block klara returnera, annars ta nästa block

Större kodexempel, initieringskod

- Initiera addresspekare (R5, R6) och sampelräknare (R4)

```

        mov r3,#0x10000    ; Antal block som ska filtreras
start:
        mov r5,#(InData & 0xffff) ; Indataadress
        movt r5,#(InData >> 16)
        mov r6,#(OutData & 0xffff) ; Outdataadress
        movt r6,#(OutData >> 16)
        mov r4,#100       ; Antal sampels

```

Filter version 1

```

        mov r3,#0x10000    ; Antal block som ska filtreras
start:
        mov r5,#(InData & 0xffff) ; Indataadress
        movt r5,#(InData >> 16)
        mov r6,#(OutData & 0xffff) ; Outdataadress
        movt r6,#(OutData >> 16)
        mov r4,#100       ; Antal sampels

inner_loop:
        adr r7,coefficients ; Koefficienter för filtret finns här
        adr r8,coefficients_end
        mov r2,#0           ; Nollställ ackumulator

sample_loop:
        ldrsh r0,[r5],#2    ; läs 16 bitar (2 byte) 2-komplement, r5=r5+2
        ldrsh r1,[r7],#2    ; Läs 16 bit koefficient
        mul r0,r1,r0        ; multiplicera (16bit x 16bit ger 32 bit resultat)
        add r2,r2,r0
        cmp r7,r8           ; Testa om använt alla koefficienter
        beq sample_ready
        b sample_loop

```

```

sample_ready:
        asr r2,r2,#15       ; Dividera med 2^15
        strh r2,[r6],#2    ; Spara 16 bit resultat i utbuffert
        sub r5,r5,#8       ; Justera a0 så den pekar på nästa
                           ; sampel med indata
        subs r4,r4,#1      ; Räkna ner loopräknaren
        bne inner_loop
        subs r3,r3,#1      ; nästa block att räkna på
        bne start

stoploop:
        b stoploop
coefficients.field -1448,16
        .field 5437,16
        .field 26559,16
        .field 5437,16
        .field -1448,16
coefficients_end
InData .equ 0x20001000
OutData .equ 0x20002000

```

Olika instruktioner tar olika lång tid

- Klockfrekvens
 - Dubbel klockfrekvens betyder inte dubbel prestanda
 - Vanligt mått: IPC (Instruction Per Clockcycle)
- Antal minnesläsningar/skrivningar
 - Kan 32 bitar läsas på en gång?
 - Hur snabbt är minnet?
- Hur lång är instruktionen (antal byte)
 - Mer eller mindre tid att hämta instruktionerna
- Villkorliga hopp kan ta olika lång tid beroende på om villkoret är sant eller falskt

Prestanda filter v1

- Exekveringstid ca 21 sekunder på Darma
 - Inte optimerat, kan göras bättre
- Vad tar onödig tid (instruktioner i loop är dyrast)
 - Vanligaste hoppet tas inte (jämför, hoppa inte, hoppa)


```

cmp    r7,r8
beqsample_ready
b    sample_loop
sample_ready:
          
```
 - Bättre lösning


```

cmp    r7,r8
bnesample_loop
sample_ready:
          
```

Filter version 2

- Vanligaste hoppet är det första som testas
- Programmet aningen mindre (2 byte kortare)
 - 1 instruktion borttagen (bra inner_loop)
- Exekveringstid
 - Ca 19 sekunder (liten förbättring)

```

inner_loop:
    adr    r7,coefficients
    adr    r8,coefficients_end
    mov    r2,#0
sample_loop:
    ldrsh  r0,[r5],#2
    ldrsh  r1,[r7],#2
    mul    r0,r1,r0
    add    r2,r2,r0
    cmp    r7,r8
    bne    sample_loop
sample_ready:
    asr    r2,#15
    strh   r2,[r6],#1
    sub    r5,r5,#8
    subs   r4,r4,#1
    bne    inner_loop
    subs   r3,r3,#1
    bne    start

stoploop:
    b      stoploop
coefficients .field: -1448,16
            .field 5437,16
            .field 26559,16
            .field 5437,16
            .field -1448,16
coefficients_end:
InData      .equ 0x20001000
OutData     .equ 0x20002000

```

Förbättringar filter version 2

- Titta på vanligaste operationerna

```

ldrsh  r0,[r5],#2
ldrsh  r1,[r7],#2
mul    r0,r1,r0
add    r2,r2,r0

cmp    r7,r8
bne    sample_loop

```

- Vad tar onödig tid
 - Test av slut på loop är "onödig" (2 av 6 instruktioner)
 - Kopiera istället instruktionerna motsvarande antal
 - Tekniken kallas Loop unrolling
 - Enklast om fixt antal varv i loop

```

inner_loop:
    adr    r7,coefficients
    adr    r8,coefficients_end
    mov    r2,#0
sample_loop:
    ldrsh  r0,[r5],#2
    ldrsh  r1,[r7],#2
    mul    r0,r1,r0
    add    r2,r2,r0
    cmp    r7,r8
    bne    sample_loop
sample_ready:
    asr    r2,#15
    strh   r2,[r6],#1
    sub    r5,r5,#8
    subs   r4,r4,#1
    bne    inner_loop
    subs   r3,r3,#1
    bne    start

stoploop:
    b      stoploop
coefficients .field: -1448,16
            .field 5437,16
            .field 26559,16
            .field 5437,16
            .field -1448,16
coefficients_end:
InData      .equ 0x20001000
OutData     .equ 0x20002000

```

Filter version 3

- Programmet växer i storlek
- Exekveringstid: 17 sekunder (10%)

```

        mov r3,#0x10000
start:   mov r5,#(InData & 0xffff)      ldrsh r0,[r5],#2
        movt r5,#(InData >> 16)        ldrsh r1,[r7],#2
        mov r6,#(OutData & 0xffff)     mul r0,r1,r0
        movt r6,#(OutData >> 16)       add r2,r2,r0
        mov r4,#100                    ldrsh r0,[r5],#2
                                           ldrsh r1,[r7],#2
                                           mul r0,r1,r0
inner_loop:
        adr r7,coefficients            ldrsh r0,[r5],#2
        adr r8,coefficients_end        ldrsh r1,[r7],#2
        mov r2,#0                      mul r0,r1,r0
sample_loop:
        ldrsh r0,[r5],#2               add r2,r2,r0
        ldrsh r1,[r7],#2               ldrsh r0,[r5],#2
        mul r0,r1,r0                   ldrsh r1,[r7],#2
        add r2,r2,r0                   mul r0,r1,r0
                                           add r2,r2,r0

```

```

sample_ready:
    asr    r2,#15
    strh   r2,[r6],#1
    sub    r5,r5,#8
    subs   r4,r4,#1
    bne    inner_loop
    subs   r3,r3,#1
    bne    start

stoploop:
    b      stoploop
coefficients.field -1448,16
    .field 5437,16
    .field 26559,16
    .field 5437,16
    .field -1448,16
coefficients_end
InData      .equ 0x20001000
OutData     .equ 0x20002000

```

Förbättring, filter version 4

- Koefficienterna alltid samma
 - Läs in dom i register innan loop
- Fungerar bäst för beräkningar med fixa värden och få konstanter
 - 5 koefficienter men bara 3 olika värden
- Ta bort konstanttabellen
 - Inre loop kortare
- Exekveringstid 13 sekunder (-23%)

Filter version 4

- Allt svårare öka prestanda
 - Det mesta av tiden spenderas på ldrsh, mul och add

```

mov r3,#0x10000
start: mov r5,#(InData & 0xffff)
movt r5,#(InData >> 16)
mov r6,#(OutData & 0xffff)
movt r6,#(OutData >> 16)
mov r4,#100

inner_loop:
mvn r7,#~(coeff1+1) ; negativt värde
mov r8,#coeff2
mov r9,#coeff3
mov r2,#0

ldrsh r0,[r5],#2
mul r0,r8,r0
add r2,r2,r0
ldrsh r0,[r5],#2
mul r0,r9,r0
add r2,r2,r0
ldrsh r0,[r5],#2
mul r0,r8,r0
add r2,r2,r0
ldrsh r0,[r5],#2
mul r0,r7,r0
add r2,r2,r0

asr r2,#15
strh r2,[r6],#1
sub r5,r5,#8
subs r4,r4,#1
bne inner_loop
subs r3,r3,#1
bne start

stoploop:
b stoploop
coeff1 .equ -1448
coeff2 .equ 5437
coeff3 .equ 26559
InData .equ 0x20001000
OutData .equ 0x20002000

```

Filter version 5

- Några saker kan förbättras
 - En instruktion som kombinerar add och mult

$$\text{mla } r2, r0, r7, r2 ; [r2, r0] = r7 * r2, \text{ dvs } 32 \text{ LSB-bitar i } r0, 32 \text{ MSB-bitar i } r2$$
- Exekveringstid: 12s

```

mov r3,#0x10000
start: mov r5,#(InData & 0xffff)
movt r5,#(InData >> 16)
mov r6,#(OutData & 0xffff)
movt r6,#(OutData >> 16)
mov r4,#100

inner_loop:
mvn r7,#~(coeff1+1)
mov r8,#coeff2
mov r9,#coeff3
mov r2,#0

ldrsh r0,[r5],#2
mla r2,r7,r0,r2
ldrsh r0,[r5],#2
mla r2,r8,r0,r2
ldrsh r0,[r5],#2
mla r2,r9,r0,r2
ldrsh r0,[r5],#2
mla r2,r8,r0,r2
ldrsh r0,[r5],#2
mla r2,r7,r0,r2

asr r2,#15
strh r2,[r6],#1
sub r5,r5,#8
subs r4,r4,#1
bne inner_loop
subs r3,r3,#1
bne start

stoploop:
b stoploop
coeff1 .equ -1448
coeff2 .equ 5437
coeff3 .equ 26559
InData .equ 0x20001000
OutData .equ 0x20002000

```

Ytterligare möjliga förbättringar

- Använd instruktion som adderar och multiplicerar två 16-bitars tal direkt
 - SMLAD rx,ry,rz,rw
 - Beräknar $rx = (ry[31:16]*rz[31:16]) + (ry[15:0]*rz[15:0]) + rw$
 - Kräver annorlunda hantering av konstanter
 - Kombinera två konstanter i ett register
 - Tex använd `movt rx,#16-bit`
 - Placera 16-bitars konstant i `rx[31:16]`
- Allt svårare hitta förbättringar
 - Allt mer processorspecifika förbättringar (svårt flytta kod till annan processorfamilj)

Andra möjligheter till snabbare exekvering

- Ytterligare förbättringar kan göras
- Kräver detaljerad kunskap om Cortex M4 och hur den utför instruktioner
 - Måste veta hur lång tid varje instruktion tar
 - Kan bero bl a på adresseringsmod
- Stort antal andra instruktioner att undersöka
 - Villkorlig exekvering
 - Bitfältsooperationer

Andra alternativ till förbättring, forts.

- Andra alternativ bygger på förändringar/byte av algoritmen
 - För den intresserade: Googla på FFA (Fast Fir Algorithm)
 - Kan vara intressant att använda t ex FFT (Fast Fourier Transform)
 - Byta multiplikation mot flera additioner
 - Exempel: $b_0=b_4 \Rightarrow b_0 \cdot x(n) + b_4 \cdot x(n-4) = b_0 \cdot (x(n) + x(n-4))$
 - Ändra beräkningsordning för att undvika läsning av samma minnesvärde flera gånger
 - Läs en gång till register och multiplicera sedan med alla olika koefficienter (resultat i olika register)

Slutsats om prestandaökning

- Går ofta att förbättra (till en viss gräns)
 - I exemplet: 21 sekunder -> 12 sekunder
 - En kraftig minskning! (40%)
- Prestandaökning inte gratis
 - Ökad mängd kod
 - Mindre flexibilitet
 - Svårt öka antal koefficienter
 - Fungerar kanske bara på en processormodell
 - Svårare förstå kod

Slutsats om prestandaökning, forts.

- En del av förbättringarna är svåra att generera automatiskt vid kompilering av högnivåspråk
 - Extrem prestanda kräver ibland assemblerprogrammering
- Kompilatorns kvalitet viktig
 - Översätter högnivåspråk till assembler
 - Som synes finns många val, kompilatorn måste vara bra på att hitta lösningar som ger snabb kod
- Nästa processorgeneration kan ha helt andra egenskaper
 - Nya instruktioner som löser delproblem bättre
 - Bredare/snabbare minnebuss
 - Fler register

Prestanda hos andra processorfamiljer

- Hastighet att utföra instruktioner endast en av flera intressanta egenskaper
 - Beror på tänkt tillämpning
 - Kostnadsmått varierar
 - Exekveringshastighet, programstorlek, stöd för datatyper och operationer
- Hastighetskrav för avbrott
 - Hur snabbt svar?
 - Extra register, automatiskt spara register etc.
 - Hur vet processorn adress för avbrott?
 - Flexibilitet vs hastighet

Andra instruktionsuppsättningar

Alternativa instruktionsuppsättningar

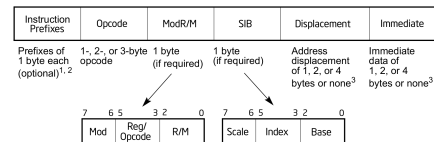
- ARM (används i ARM Cortex-A, t ex Raspberry Pi)
 - Möjliggör villkorlig exekvering av flytt, aritmetiska och subrutinhoppsoperationer
 - CMP r0, #0 ; if (x <= 0)
 - MOVLE r0, #0 ; x = 0;
 - MOVGT r0, #1 ; else x = 1;
 - Tillåter val om flaggor ska påverkas av en instruktion
 - Varje instruktion är 32 bitar lång
 - Väldigt få adresseringsmoder
- Cortex-A familjen kan tolka olika instruktionsuppsättningar
 - ARM vs THUMB
 - Olika längd på instruktioner beroende på instruktionsuppsättning

Alternativa instruktionsuppsättningar, forts.

- Processorn tolkar binära data som instruktioner (maskininstruktioner)
- Mappning instruktion => binärmönster
 - Kompatibilitet kan vara binär
 - samma binärmönster => samma beteende
 - Kompatibilitet kan vara på källkods nivå
 - Måste assemblera om källkod för att kunna köras på ny maskin
- Mappningen måste vara effektiv att implementera
 - Inte något slumpmässigt val av bitmönster
 - Ofta möjligt dela upp instruktionen i olika bitfält
 - Ett för opcode, ett för register, ett för konstantargument etc.
 - Vill inte ha oanvända bitar

Alternativa instruktionsuppsättningar: x86

- Intel x86 (vanliga windows/MacOS datorer)
- Instruktioner är 8-120 bitar
 - 0x43 => INC BX ; BX = BX + 1
 - 0x32 0xf1 0x12 => XOR CL,0x12
 - 0x8B 0x88 0x45 0x03 => MOV CX,COUNT[BX][SI]
- Prefix för att utöka antal möjliga instruktioner
 - Exempel: REPE Repeat while equal
- Max 15 byte för en instruktion
 - Varje ruta motsvarar ett antal bitar som väljer en funktion
 - Reg : vilket register (finns bara 8 generella)
 - Mod : om register eller fler byte i instruktionen ska ingå
- 2 operand instruktioner (ett av indataregistren är utdataregister)



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" for additional information.
 2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)".
 3. Some rare instructions can take an 8B immediate or 8B displacement.

<https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html>

Alternativa instruktionsuppsättningar: ARM

- ARM (t ex Cortex-A), OBS: Ej thumb (dvs inte Dharma)
 - 32-bitar långa instruktioner (varken mer eller mindre)
 - Adresser och konstanter kan behöva vara 32 bitar
 - Kan inte ingå i instruktionen direkt
 - Alternativ 1: kombineras MOV och MOVT
 - Alternativ 2: kort konstant gör om till lång
 - Alternativ 3: Ladda konstant från programminne (adress < 12 bitar)
 - Instruktionen behöver beskriva vilken operation, vilka register, vilka övriga parametrar
 - Bit 31-26 anger villkor (Alla instruktion är villkorliga)
 - Bit 27 och 26 beskriver vilken typ av instruktion
 - 00 databehandlande
 - 01 load/store
 - 10 hopp/subrutiner blockflytt
 - 11 coprocessor, software interrupt

Alternativa instruktionsuppsättningar: ARM

- ARM instruktioner delas upp på olika fält

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition	0	0	1																													Data processing
Condition	0	0	0	0	0	0	A	S																								Multiply
Condition	0	0	0	1	U	A	S																									Long Multiply
Condition	0	0	0	1	0	B	0																									Swap
Condition	0	1	1	P	U	B	W	L																								Load/Store - Byte/Word
Condition	1	0	0	P	U	B	W	L																								Load/Store Multiple
Condition	0	0	0	P	U	1	W	L																								Halfword Transfer Imm Off
Condition	0	0	0	P	U	0	W	L																								Halfword Transfer Reg Off
Condition	1	0	1	L																												Branch
Condition	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1						Branch Exchange
Condition	1	1	0	P	U	N	W	L																								COPROCESSOR DATA XFER
Condition	1	1	1	0																												COPROCESSOR DATA OP
Condition																																COPROCESSOR REG XFER
Condition	1	1	1	1																												Software Interrupt

https://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf

- THUMB kombinerar 16 och 32 bitar långa instruktioner
 - Måste förenkla jämfört med vanlig ARM
 - Inget condition fält
 - Kortare offset/konstanter
 - Vill fortfarande kunna köra nästan likadana program
 - Ovanligare operation blir 32 bitar långa

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate, move register	0	0	0	opcode [1]			imm5					Rm		Rd		
Add/subtract register	0	0	0	1	1	0	opc		Rm		Rn		Rd			
Add/subtract immediate	0	0	0	1	1	1	opc		imm3		Rn		Rd			
Add/subtract/compare/move immediate	0	0	1	opcode			Rdn		imm8							
Data-processing register	0	1	0	0	0	0	opcode			Rm		Rdn				
Special data processing	0	1	0	0	0	1	opcode [1]		DN	Rm		Rdn				
Branch/exchange instruction set [3]	0	1	0	0	0	1	1	1	L	Rm		(0)		(0)	(0)	
Load from literal pool	0	1	0	0	1	Rd			PC-relative imm8							
Load/store register offset	0	1	0	1	opcode			Rm		Rn		Rd				
Load/store word/byte immediate offset	0	1	1	B	L	imm5			Rn		Rd					
Load/store halfword immediate offset	1	0	0	0	L	imm5			Rn		Rd					
Load from or store to stack	1	0	0	1	L	Rd		SP-relative imm8								
Add to SP or PC	1	0	1	0	SP	Rd		imm8								
Miscellaneous: See Figure 6-2	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	
Load/store multiple	1	1	0	0	L	Rn		register list								
Conditional branch	1	1	0	1	cond [2]			imm8								
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	
Service (system) call	1	1	0	1	1	1	1	1	imm8							
Unconditional branch	1	1	1	0	0	imm11										
32-bit instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	
32-bit instruction	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	

ARM Architecture Reference Manual Thumb-2 Supplement

Prestandajämförelser

Prestandajämförelser

- Antal register, operationstyper, klockfrekvenser och ordbredder skiljer sig åt
 - Varierar även inom samma familj av processorer
- Köparen av en dator är intresserad av prestanda (tid, effekt, kostnad) för att köra köparens applikationer
 - Olika processorer är bra på olika saker (heltalsaritmetik, sekvensiell vs parallel exekvering etc.)
- Vill därför kunna jämföra prestanda för vissa typer av arbetsuppgifter

Prestandajämförelser, forts.

- Standardiserade test: SPEC
 - Standard Performance Evaluation Corporation
- SPEC CPU2017
 - SPECrate/SPECspeed 2017 Integer: 10 olika program att mäta heltalsprestanda på (inkl. matematiska beräkningar med heltal/2-komplement)
 - SPECrate/SPECspeed 2017 Floating point: 13/10 olika program att mäta flyttalsprestanda på (matematiska beräkningar med reella tal)
- Definierade test ger samma effekt som för dieslbilars utsläppstester.....
 - Processorer blir optimerade för att ge bra testresultat
 - Bra testresultat stämmer inte alltid med bra prestanda för önskad applikation.
- Vanligt med stor mängd "verkliga" testfall istället

Prestandajämförelser, forts.

- Alternativa jämförelser
 - Generella, sammanfattande
 - PC Mark, SISoftware Sandra
 - Specifika, vanliga "nyttoprogram"
 - Office
 - Adobe
 - Zip
 - Specifika, beräkningsserver
 - NAMD, Unixbench, sysbench
 - Specifika, Gaming
 - Battlefield, hitman, GTA etc.

Prestandajämförelser, forts.

- Att komma ihåg vid jämförelse av datorer
 - Processorn är en del av ett större system
 - Prestanda kan begränsas av många andra delar
 - Hårddisk vs SSD
 - Minne
 - Kylning
 - Prestanda påverkas även av kompilatorns funktion
 - Intel har egen assembler, ARM har egen assembler
 - gcc, llvm open source alternativ

Nästa period (VT2)

- Dags att lära sig hur datorn är uppbyggd i detalj
 - Introducera bussar, register, beräkningselement, klockcykler, styrsignaler etc.
 - Se hur arkitektur bestämmer hastighet hos datorn
- Därefter se hur prestanda kan ökas mha diverse lösningar
 - Förutse nästa instruktion
 - Använda små snabba minnen
 - Göra flera saker samtidigt

Nästa period (VT2), forts.

- Ingen mer ARM Cortex-M assembler i nästa period
 - Undantag: Tenta!
 - Titta på gamla tentor för att se upplägg och tillgänglig information
 - Se till att träna så ni kan skriva egna små program
 - Träna att analysera given kod
- Förväntar mig inte att ni kan alla instruktioner
 - Ska kunna de instruktioner ni använde i labben
 - Många specialinstruktioner är inte nödvändiga
 - Darna-manualen ger en bra lista

