TSEA28 Datorteknik Y (och U)

Föreläsning 16

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Mer avancerade sätt att öka prestanda
- Applikationsspecifika processorer
- Profilering
- Sammanfattning
- Kommentarer om tentan

Praktiska kommentarer

- Access till Grinden (lab5) via liu-kort
  - Stängt 24/5 – 26/5, 7/6 pga LAX i annan kurs
- Fokusera på lab5 vid schemalagd lab
  - Lås inte upp labassistenter med lab4!
- Extra labbredovisningsmöjligheter
  - Lab 1-3 i MUX1 Torsdag 8/6 kl 9.15
  - Lab 4 i MUX1 Fredag 9/6 kl 9.15
  - Lab 5 i Grinden Fredag 9/6 kl 13
- Labutrustning (lab 1-3, 5) kommer monteras ned efter redovisningstillfällena ovan
  - Lab 1-3 redan nedmonterat med undantag för distansutrustning i mux2

Årets vinnare i sorteringstävlingen

- Vinst för vinnaren: Ett paket ballerinakex
- Tävlingsresultat (medel av 5 slumpmässiga sekvenser)
  - ~~20951~~ (fel sorteringsresultat)
  - 7558
  - 4185
  - 3275
  - 2067
  - 1832
  - 1709

Årets vinnare av ballerinapaket

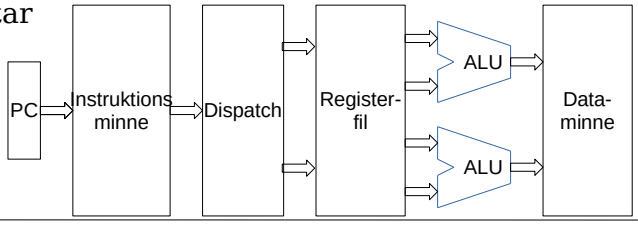
- Herman Lauenstein och Alexander Berntsson: 1709.6
- Som jämförelse, bästa resultat tidigare år:
  - 2014: 1119.6
  - 2016: 2216
  - 2017: 2349
  - 2018: 1337
  - 2019: 1016.8
  - 2020: 1630.8
  - 2021: 4294
  - 2022: 1562.2

Sedan tidigare

- Pipelining tillåter flera instruktioner utförda omlott
  - Tillåter fler startade instruktioner per tidsenhet
- Superskalär processor tillåter flera instruktioner startade samtidigt (samma klockcykel)
  - Tillåter ännu fler instruktioner per sekund
- Problem med konflikter (hazards)
  - Data (resultat inte färdigt före användning)
  - Styr (hopp, villkorliga hopp)
  - Strukturell (ALU används av flera instruktioner samtidigt)

Superskalär processor, arkitektur

- Instruktionsminne skickar flera instruktioner per läsning
- Dispatch fördelar instruktioner till beräkningsenheter
- Registerfil med fler läs och skrivportar
- Flera exekveringsenheter
- Eventuellt minne med flera vägar
  - T ex datacache plus skrivbuffertar



Problem och dellösningar för superskalär pipelined RISC

- Datakonflikt
  - Forwarding (kopiera ALU utdata till ALU indata)
- Styrkonflikt
  - Branch prediction (gissa nästa instruktionsadress)
- Strukturell konflikt
  - Dispatch delar ut instruktioner att utföra mellan de två ALU och minnesvägarna
- Kräver fortfarande anpassning av kod för att få snabbare exekvering
  - Ändring av antal och typ av beräkningsblock kräver annorlunda kod (instruktionsordning)
  - Löser inte alla prestandaproblem

Exempel konflikthantering (2-vägs superskalär processor, 1 minnesport)

- Orginalkod: skalärprodukt på två vektorer med 40 element var
- Orginalkod och utrullad kod (2 varv) $\sum_{i=1}^{40} x_i \cdot y_i = x_1 \cdot y_1 + x_2 \cdot y_2 + ... + x_{40} \cdot y_{40}$

```
dotprod:    mov  r0,#40
            mov  r1,#0
loop:
        {   ldrsh r2,[r4],#2
        {   ldrsh r3,[r5],#2
        {   mul   r3,r3,r2
        {   add   r1,r1,r3
        {   sub   r0,r0,#1
            bne   loop
            bx    lr
```

```
dotprod:    mov  r0,#40,     ; kan köras samtidigt med mov r1,#0
            mov  r1,#0
loop:
        {   ldrsh r2,[r4],#2 ; 1 minne, kan inte köras samtidigt med nästa
        {   ldrsh r3,[r5],#2 ; -   '   '   -
        {   ldrsh r6,[r4],#2 ; -   '   '   -
        {   ldrsh r7,[r5],#2 ; kan köras samtidigt med mul
        {   mul   r3,r3,r2
        {   mul   r7,r7,r6   ; kan köras samtidig med add
        {   add   r1,r1,r3
        {   add   r1,r1,r7   ;
        {   subs  r0,r0,#2   ; Räkna ner loopräknaren med TVÅ!
            bne   loop
            bx    lr
```


Mycket outnyttjad parallellism tillgänglig

- Orginalkod: 2 minnesaccess + 2 beräkningar per varv (+ 1 beräkning och 1 hopp)
- Antag 2-vägs superskalär!
  - Som bäst behövs bara 2 klockcykler per varv om loopstyrning kan undvikas
    - 2 minnesläsningar + 2 beräkningar

```
dotprod2t:  mov  r0,#40      ; kan köras samtidigt med mov r1,#0
            mov  r1,#0
loop2t:
        {   ldrsh r2,[r4],#2 ; 1 minne, kan inte köras samtidigt med nästa
        {   ldrsh r3,[r5],#2 ; -   '   '   -
        {   ldrsh r6,[r4],#2 ; -   '   '   -
        {   ldrsh r7,[r5],#2 ; kan köras samtidigt med mul
        {   mul   r3,r3,r2
        {   mul   r7,r7,r6   ; kan köras samtidig med add
        {   add   r1,r1,r3
        {   add   r1,r1,r7   ;
        {   subs  r0,r0,#2   ; Räkna ner loopräknaren med TVÅ!
            bne   loop2t
            bx    lr
```

Bättre implementering: Fler varv

- Två parallella 2 varvs utrullning (antag det finns fler register)
- Flytta ordning, räkna inte ihop slutsumman direkt (ingen delayslot i detta exempel)
  - Många minnesläsningar i sekvens etc.

```
dotprod4t:  mov  r0,#40
            mov  r1,#0
            mov  r12,#0
loop4t:
            ldrsh r2,[r4],#2
            ldrsh r3,[r5],#2
            ldrsh r6,[r4],#2
            ldrsh r7,[r5],#2
            ldrsh r8,[r4],#2
            ldrsh r9,[r5],#2
            ldrsh r10,[r4],#2
            ldrsh r11,[r5],#2
            mul   r3,r3,r2
            mul   r7,r7,r6
            mul   r9,r9,r8
            mul   r11,r11,r10
            add   r1,r1,r3
            add   r12,r12,r7
            add   r1,r1,r9
            add   r12,r12,r11

            subs  r0,r0,#4
            bne   loop4t

            add   r1,r1,r12
            bx    lr
```


Bättre implementering: Fler varv, forts.

- Flyttat hälften av minnesläsningarna sist i loop
  - Måste då även sätta rätt värden innan loop startar

```
dotprod4t:  mov r0,#40
            mov r1,#0
            mov r12,#0
            ldrshr2,[r4],#2
            ldrshr3,[r5],#2
            ldrshr6,[r4],#2
            ldrshr7,[r5],#2
Loop4t:
            ldrshr8,[r4],#2
            ldrshr9,[r5],#2
            ldrshr10,[r4],#2
            ldrshr11,[r5],#2
            mul  r3,r3,r2
            mul  r7,r7,r6
            mul  r9,r9,r8
            mul  r11,r11,r10
            add  r1,r1,r3
            add  r12,r12,r7
            add  r1,r1,r9
            add  r12,r12,r11
            ldrshr2,[r4],#2
            ldrshr3,[r5],#2
            ldrshr6,[r4],#2
            ldrshr7,[r5],#2
            subsr0,r0,#4
            bne  loop4t
            add  r1,r1,r12
            bx   lr
```

Risk med flyttade minneläsningar

- 4 data läses extra (utanför vektorerna)!
  - Kan ge försök att läsa otillåtna minnesadresser (minnesskydd?)
- Löses enklast med extra specialavslutning av loop
  - Prolog före loop
    - Läs indata
  - Epilog efter loop
    - Avslutande skrivningar

```
dotprod4t: mov  r0,#40          mul  r9,r9,r8
           mov  r1,#0           mul  r11,r11,r10
           mov  r12,#0          add  r1,r1,r3
           ldrsh r2,[r4],#2     add  r12,r12,r7
           ldrsh r3,[r5],#2     add  r1,r1,r9
           ldrsh r6,[r4],#2     add  r12,r12,r11
           ldrsh r7,[r5],#2     ldrsh r2,[r4],#2
Loop4t:                         ldrsh r3,[r5],#2
           ldrsh r8,[r4],#2     ldrsh r6,[r4],#2
           ldrsh r9,[r5],#2     ldrsh r7,[r5],#2
           ldrsh r10,[r4],#2    subs r0,r0,#4
           ldrsh r11,[r5],#2    bne  loop4t
           mul  r3,r3,r2        add  r1,r1,r12
           mul  r7,r7,r6        bx   lr
```


Nästan nere på 2 klockcykler per varv (9 klockcykler / 4 varv)

- Sprid ut beräkningarna bland läsinstruktionerna
  - Tillåter samtidigt beräkning och läsning utan databeroende
- Denna typ av omskrivning känd som software pipelining
- För en VLIW-processor skulle programmeraren behöva utföra denna omskrivning

```
dotprod4t: mov  r0,#40
           mov  r1,#0
           mov  r12,#0
           ldrsh r2,[r4],#2     { ldrsh r2,[r4],#2
           ldrsh r3,[r5],#2     { mul  r9,r9,r8
           ldrsh r6,[r4],#2     { ldrsh r3,[r5],#2
           ldrsh r7,[r5],#2     { mul  r11,r11,r10
Loop4t:                         { ldrsh r6,[r4],#2
           ldrsh r8,[r4],#2     { add  r1,r1,r9
           mul  r3,r3,r2        { ldrsh r7,[r5],#2
           ldrsh r9,[r5],#2     { add  r12,r12,r11
           mul  r7,r7,r6        { subs r0,r0,#4
           ldrsh r10,[r4],#2    { bne  loop4t
           add  r1,r1,r3
           ldrsh r11,[r5],#2      add  r1,r1,r12
           add  r12,r12,r7        bx   lr
```

Automatiserat sätt att ändra instruktions-ordning

- Processorn hämtar in många instruktioner och avkodar dom
- Bestäm sedan vilka som ska utföras beroende på om indata finns tillgängligt
  - Instruktioner kan då utföras i annan ordning
  - Måste garantera att slutresultat fortfarande blir samma
  - I exemplet: Läs flera varv ur orginalkod, välj att exekvera instruktioner som har indata tillgängligt i respektive klockcykel
- Känt som out-of-order execution
  - Kräver mycket hårdvara och bra branch prediction

Exempel på superskalär exekvering, avancerad version

- Samma exempel som Figure 8.6 – Figure 8.7 i boken

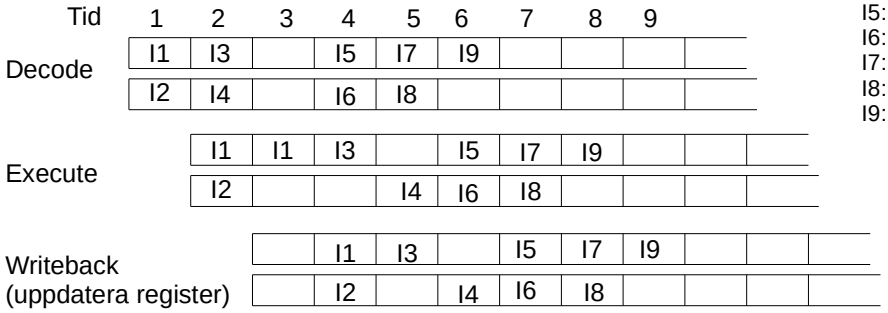
```
I1: ldr   r0,[r10]     ; r0 = minne(r10)
I2: add   r10,r10,#4   ; r10 = r10 + 4
I3: mul   r3,r2,r1     ; r3 = r1*r2
I4: mul   r4,r1,r0     ; r4 = r1*r0
I5: sub   r5,r4,r6     ; r5 = r4 – r6 , r4 skapad i I4:
I6: add   r7,r8,r2     ; r7 = r8 + r2
I7: mul   r0,r9,r5     ; r0 = r9*r5
I8: add   r1,r1,#4   ; r1 = r1 + 4
I9: add   r7,r7,#1   ; r7 = r7 + 1
```

- Antag 2-väg superskalär processor med 2 klockcyklers minnesläsning, 1 multiplikator.

Exempel på superskalär processor, forts.

- 2 klockcykler för minnesaccess, 1 mult.

```
I1:  ldr  r0,[r10]
I2:  add  r10,r10,#4
I3:  mul  r3,r2,r1
I4:  mul  r4,r1,r0
I5:  sub  r5,r4,r6
I6:  add  r7,r8,r2
I7:  mul  r0,r9,r5
I8:  add  r1,r1,#4
I9:  add  r7,r7,#1
```

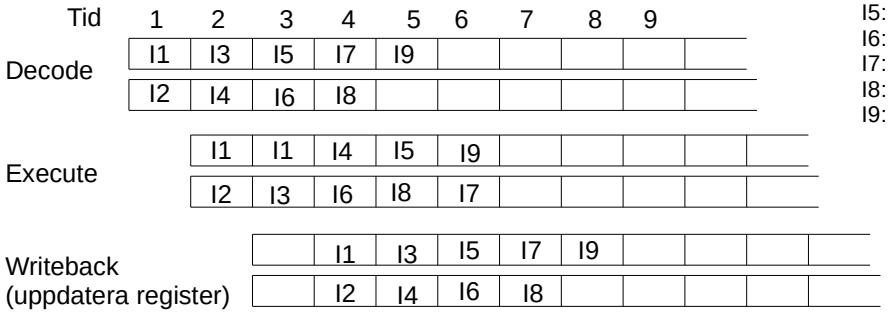


Samma ordning på registeruppdatering som enligt koden.
Begränsning: 2 cykel minnesläsning, 1 multiplikator

Exempel på superskalär processor, out-of-order exekvering

- 2 klockcykler för minnesaccess, 1 mult.

```
I1:  ldr  r0,[r10]
I2:  add  r10,r10,#4
I3:  mul  r3,r2,r1
I4:  mul  r4,r1,r0
I5:  sub  r5,r4,r6
I6:  add  r7,r8,r2
I7:  mul  r0,r9,r5
I8:  add  r1,r1,#4
I9:  add  r7,r7,#1
```



Begränsning: 2 cykel minnesläsning, 1 multiplikator
I6 utförs nu före I5, och I8 före I7. Dock samma ordning på registeruppdatering!

Exempel på register renaming

- Exempel: register r0 återanvänds (WAR konflikt)

```
I1: ldr r0,[r4]      ; r0 = minne(r4)
I2: add r2,r2,r0     ; r2 = r2 + r0
I3: ldr r0,[r5]      ; r0 = minne(r5)
I4: add r3,r3,r0     ; r3 = r3 + r0
```

- Register r0 inte tillgängligt för instruktion I3, måste vänta tills I2 halvvägs klar om pipelined/superskalär processor
  - Alternativ (snabbare) kod

```
I1: ldr r0,[r4]      ; r0 = minne(r4)
I2: add r2,r2,r0     ; r2 = r2 + r0
I3: ldr r1,[r5]      ; r1 = minne(r5)
I4: add r3,r3,r1     ; r3 = r3 + r1
```

- Registerval ger extra begränsningar

Ytterligare aktiviteter

- Out-of-order execution
  - Kan även tillåta out-of-order issue (start) och out-of-order completion (avslutning)
  - Låt processorn själv välja instruktionsordning (och registeruppdateringsordning)
  - Så länge som slutresultat är korrekt kvittar ordningen
- Register renaming
  - Starta instruktioner även om resultatregistret i operationskoden inte är tillgängligt ännu
  - Använder fler register än vad programmeraren känner till
- Spekulativ exekvering
  - Utför instruktioner även om det ännu inte är klart att de behöver utföras
    - T ex pga oklar branch prediction (utför båda vägarnas operationer)
  - Spara resultatet internt (skriv inte ut till minne/register)
  - Mer aggressivt: Räkna utan att veta indata (antag t ex indata=0)

CISC vs RISC instruktioner, strängjämförelse

- CISC har komplexa instruktion som kan kräva många klockcykler
  - Exempel x86: (cx=max antal, si=adress sträng 1, di=adress sträng 2)

```
REPNE CMPSB   ; Jämför 2 strängar tills olika eller max tecken testats
```

  - Motsvarande i ARM (R3=max antal, R0 = adress sträng 1, R1 = adress sträng 2)

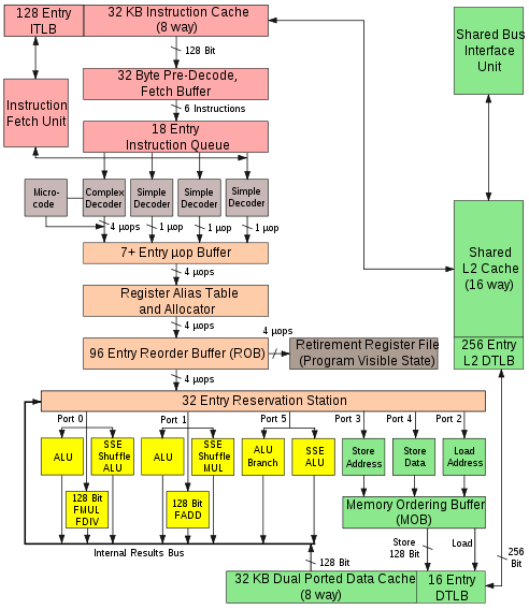
```
Loop:
    LDRB R5, [R0], #1     ;  Tecken i sträng 1, räknar upp R0 med 1
    LDRB R6, [R1], #1     ;  Tecken i sträng 2, räknar upp R1 med 1
    CMP  R5, R6           ;  Är dom lika?
    BNE  Done             ;  Skillnad, klar med jämförelse, Z-flaggan=0
    SUBS R3, R3, #1       ;  Är alla tecken jämförda?
    BEQ  Done             ;  Ja, klar med jämförelse, Z-flaggan=1
    B    Loop             ;  Inte klar, loop
Done:
```


CISC vs RISC instruktioner, öka prestanda

- CISC har kompakt kod, men detta omöjliggör superskalär implementation
  - Olika längd på instruktioner, olika antal klockcykler beroende på indata etc.
- Lösning: Översätt till RISC-kod
  - Metod 1: emulering
    - Interpretator (i RISC-kod) simulerar CISC-instruktionerna
  - Metod 2: statisk översättning
    - Statisk översättning av CISC maskinkod till RISC-kod innan körning
    - Försök optimera översatt kod
  - Metod 3: dynamisk översättning
    - Översätt under körning, analysera exekverade RISC-koden och förbättra översättningen.

Exekvering av CISC-kod i en RISC

- CISC har långa komplexa instruktioner
  - Komplicerat göra superskalära processorer av CISC-typ
  - Svårt veta var nästa instruktion börjar
- Översätt maskininstruktioner hos CISC till RISC-kod först (medan programmet körs)
  - Kör sedan RISC-instruktionerna i en ”riktig” RISC
- Moderna PC-maskiner (Intel, AMD) använder emulering
  - Ett extra block mellan FETCH och DECODE
  - Ökar kostnad för missad branch prediction



Intel Core 2 Architecture

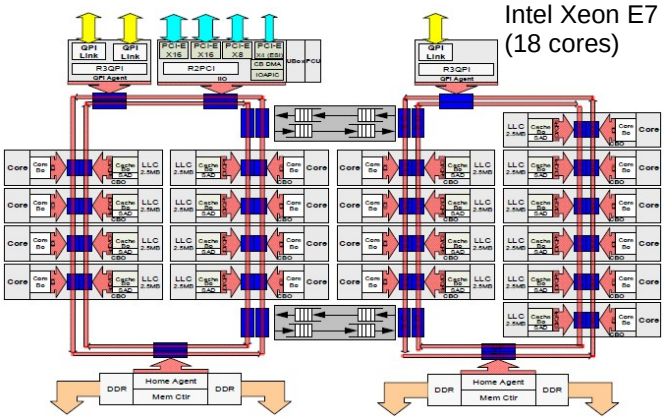
https://upload.wikimedia.org/wikipedia/commons/6/60/Intel_Core2_arch.svg

Svårt ge hålla beräknings-enheter aktiva

- Superskalära processor hittar ofta mellan 1 – 2 instruktioner per klockcykel
  - Cache-missar, branch prediction missar, etc.
- Alternativ: Kör två olika program samtidigt på processorn
  - Varje program behöver egen uppsättning register (inklusive programräknare och flaggor)
  - Kan dela på instruktionsavkodning, beräkningsenheter etc.
  - De flesta datorer kör redan flera processer parallellt med hjälp av operativsystemet
- Känt som hyperthreading eller SMT (Symmetric Multi Threading)
  - Istället för att stanna när 1:a tråden stannar pga cachemiss så kör 2:a tråden fler instruktioner istället
  - Ökar risken för strukturella konflikter

Ännu mera parallellism

- Kombinera flera processorer i samma krets
  - Kräver att uppgiften kan delas upp i flera mer eller mindre oberoende delar
- Kommunikation mellan processorer?
  - Minnesarea?
  - Direkt länk? (Message passing)
- Olika cache (närmast processorn)
  - Hur garantera samma innehåll i alla cache hos alla processorer?



http://www.nextplatform.com/2015/05/05/intel-puts-more-compute-behind-xeon-e7-big-memory/

Ytterligare sätt att öka prestanda

- Acceleratorer
  - Processorn säger till en annan enhet att göra beräkningarna
  - MPEG avkodare, AES kryptering, etc.
- Kompaktare kodformat/flera maskininstruktionsuppsättningar
  - ARM: Thumb mode (16-bitars maskininstruktioner) vs Arm ISA (32-bitars maskininstruktioner)
- Energieffektivitet
  - Sänk/öka klockfrekvens efter behov
  - Olika komplicerade processorstrukturer körs beroende på prestandakrav
    - Enklare processorer drar mindre energi per operation
    - Mer komplicerade processorer ger högre presetanda men drar mer energi per operation
    - Big.LITTLE hos ARM och Intel (byt under körning processor när beräkningskrav ändras)

Applikationsspecifika processorer (ASIP)

- Bygg en processor helt anpassad för en speciell applikation
  - Lägg till specialinstruktioner som är vanliga
  - Lägg till extra funktionsenheter för att ge bättre prestanda
  - Ta bort instruktioner som inte behövs
- I skalärproduktsexemplet
  - Speciell loopinstrukion
  - Speciell Multiplicera-Addera (MAC) instruktion
  - Dubbla minnesportar

```
dotproduct:   clr.l d1
              repeat 40, endloop
              mac (a0)+,(a1)+,d1
endloop:      rts
```


Profilering

- Prestandaanlys av program
  - Ta reda på vilken del i programmet som tar mest tid
  - Räcker inte titta på antal instruktioner i varje rutin
    - T ex: Cachemissar syns inte (jfr lab 5 uppgift 1)
- En metod: Sampla programräknaren med jämna mellanrum
  - Avbrottsrutin sparar PC-värde i en lista
  - Rutiner som ofta är med i listan behöver analyseras och optimeras
  - Kräver bara en timer som kan avbryta

Profiling, exempel under linux

- Låt gcc kompilera med flaggan -pg
- Kör programmet som vanligt
- Kör gprof på programmet för att få statistik
- Exempel från mplayer med H.264-fil

```
Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call name
12.40      2.27     2.27 73363735     0.00     0.00 decode_residual
11.86      4.44     2.17  6180354     0.00     0.00 put_h264_qpel8o
 8.14      5.93     1.49 21857226     0.00     0.00 h264_h_loop_fil
 7.43      7.29     1.36  9922560     0.00     0.00 ff_h264_decode_
 5.52      8.30     1.01 18181724     0.00     0.00 put_h264_chroma
 4.70      9.16     0.86    82688     0.01     0.07 loop_filter
 4.67     10.02     0.86  9922560     0.00     0.00 ff_h264_filter_
 4.43     10.83     0.81 23096042     0.00     0.00 h264_h_loop_fil
```


Avancerad profilering

- Moderna processorer har ofta extra stöd för profilering
  - Avbrott efter n stycken cachemissar, branch-prediction missar etc.
  - Spara ner PC i dessa fall istället så kan du profilera ditt program för att se vilka delar som ger cachemissar
    - linux: OProfile
    - Windows: Vtune, Visual Studio ska också stödja detta
- Även hårdvarustöd för debug är vanligt
  - Trace buffer
  - Hårdvarustöd för brytpunkter (Darma har detta)

Summering av kursen

- En del nyckelord
  - Maskinkod, assemblerkod, addresseringmoder
  - Register, flaggor, minne, ALU
  - Stack, subrutiner, hopp, villkorliga hopp
  - Binär aritmetik, tvåkomplement, hexadecimala tal
  - I/O, avbrott, prioritet
  - ARM Cortex-M grunder
  - Virtuellt minne, minnesskydd

Summering av kursen, forts.

- Fler nyckelord
  - Buss, tristate-grind, register, multiplexer
  - Mikroprogrammering, styrsignaler
  - CISC, RISC, pipeline,
  - von Neumann, Harvard, DMA
  - Minnestyper, cacheminne, associativitet, tag
  - Konflikter (hazard), forwarding, branch prediction
  - DSP, SIMD, VLIW, Superskalär, hyperthreading

Kommentarer om tentan

- Bygger på material från föreläsningar, labbar och kursbok/kursböcker (kursmaterial)
- Samma grundstruktur som tidigare tentor
  - Medskick av datormodell och ARM Cortex-M instruktioner
  - Ska både kunna analysera och skriva enkla assemblerprogram och mikroprogram
- Läs framsidan på tentaomslaget!
  - Ingen rödpenna
  - Endast en uppgift per papper (flera delproblem på samma papper är ok)
  - Skriv läsligt

Kommentarer om tenta, forts.

- Skriv kommentarer till eventuell kod du skriver!
  - Kan få några poäng även om fel instruktion används
- Rita gärna figurer för att förklara
- Ingen speciell ordning på uppgifter
  - Inte ordnat efter svårighet eller liknande
- Redovisa alla relevanta mellansteg i eventuella uträkningar

Framåtblick

- Andra kurser som kan vara intressanta
  - TSEA56 Elektronik kandidatprojekt
  - TSEA81 Datorteknik och Realtidssystem
  - TSEA26 Konstruktion av inbyggda DSP processorer
  - TSEA44 Datorteknik – ett datorsystem på ett chip
  - IDA:s kurs om Datorarkitektur
  - IDA:s kurs om parallellprogrammering och multicore
  - IDA:s kurs om kompilatorer
  - GPU-programmering…
  - ….

LINKÖPINGS UNIVERSITET