EDA 332/DIT122 Datorsystemteknik

11:e Oktober 2019

1. (a) Enligt konventionen innehåller $a0 och $a1 inparametrarna, medan resultatet hamnar i $v0. I kommentarerna kallar vi $a0 för u och $a1 för j.

```
        addi    $t0, $zero, 0       # i = 0
        addi    $t1, $zero, 0       # tmp = 0
L1:     sll     $t2, $t0, 2         # $t2 = i * 4
        add     $t3, $a0,$t2        # $t3 = u + 4*i, dvs &u[i]
        lw      $t4, 0($t3)         # $t4 = u[i]
        add     $t1, $t1, $t4       # tmp += u[i]
        addi    $t0, $t0,1          # i++
        slt     $t5, $t0, $a1       # if (i < j)
        bne     $t5, $zero, L1      # goto L1
        add     $v0, $zero, $t1     # flytta tmp till utregistret
```

Koden beräknar summan av u[i] där i går mellan 0..j

(b) i. En byte anländer var tjugonde mikrosekund, dvs en gång per 20000 cykler. Det tar 200 cykler att hantera avbrottet och sedan 1000 cykler för att läsa in byten och spara den: summa 1200 cykler. Då åtgår 6% (1200/20000) av kapaciteten för denna syssla.

ii. Samma svarstid som i avbrottsfallet garanteras om vi i varje 200-cykel-period spenderar 75 cykler på en pollningsoperation. När pollningsoperationen hittar data (var tjugonde mikrosekund) kommer dessutom 1000 cykler att spenderas på att ta hand om datat. Då åtgår $1000+(75/200)\cdot(20000-1000) = 8125$ cykler, eller 8125/20000 = 40.6% av den totala kapaciteten, för att hantera denna I/O.

2. (a) Större blockstorlek utnyttjar rumslokaliteten bättre.
(b) Dock: blir blocken alltför stora jämfört med hela cachens storlek ökar antalet konfliktmissar, eftersom en större del av cachens innehåll kastas ut vid varje miss. Med blockstoleken 256B rymmer cachen bara 4 block!

(c) Den genomsnittliga misskostnaden per minnesåtkomst ges av miss-sannolikheten och kostnaden per miss. För de fyra fallen har vi:

4 B: $0.38 \cdot (4 + 4/4) = 1.9$
16 B: $0.22 \cdot (4 + 16/4) = 1.76$
64 B: $0.19 \cdot (4 + 64/4) = 3.8$
256 B: $0.27 \cdot (4 + 256/4) = 18.36$

Vi ser att blockstorleken 16 bytes är att föredra.
[Vi kan också observera dels att lägsta miss-rate inte ger bäst prestanda, dels att inte ens en mycket låg miss rate för blockstorleken 256 bytes skulle gjort detta parameterval konkurrenskraftigt.]

3 (a). Antal block i cache = **S/B**.

(b). Antal sets i cache = Ant. block / associativitet = **S/(B*A)**.

c) Addressen består av tag,index, och offset. Offset behöver **b** bitar. Varje set har ett eget index så antalet indexbitar är **log2 (S/(B*A),** vilket även kan skrivas som **s-b-a.** Totala antalet bitar är **W** och resterande bitar utgörs av tagfältet. Antal tagbitar är W-(s-b-a)-b = W-s+b+a-b = **W-s+a.**

d) För varje block har vi data,tag,Valid-bit, och en Dirty-bit (vi behöver inga R-bitar eftersom det är Random som algoritm). Data använder **B** bytes=**8B** bitar. Tagfältet använder **W-s+a** bitar. Valid- och Dirty-bitar kräver 2 bitar. Totalt per block krävs **8B+W-s+a+2**, och för hela cachen då **S/B * (8B + W – s + a + 2)** bitar.

e) Om cache access och addressöversättning skall kunna ske samtidigt så måste sidoffset (den del som inte översätts) vara tillräckligt stor för att kunna representera index och blockoffset i cacheaddresserna. C-uppgiften ovan gav oss antalet bitar för index och blockoffset= s-b-a +b = s-a. Så sidoffset måste minst alltså vara s-a bitar vilket betyder att sidorna måste vara större än 2^{s-a} =**S/A** bytes.

4. a. CPU-tid = I * CPI * Tc

I påverkas av ISA, program samt kompilator

CPI påverkas av ISA, program, kompilator, processoruppbyggnad, minnesorganisation

Tc påverkas av processoruppbyggnad, hårdvaruteknologi

b. Kan användas då den instruktion som behöver data från en tidigare instruktion tidigast behöver dem i samma cykel som de produceras.

c. Om instruktionen som behöver data ska använda dem i samma cykel som de produceras, och om de blir tillgängliga först en bit in i cykeln, så kan det krävas att cykeltiden förlängs för att hinna med. En förlängning av cykeltiden slår på alla instruktioner, och det är därför stor risk att denna negativa effekt är större än den positiva effekten av att undvika vissa datakonflikter.

d. Ersätt additionerna på rad 9-10 med multiplikation med 4 (vänsterskifta 2 steg). Ersätt nop i delay branch luckan på rad 12 med multiplikationen med 4. Sätt $a2 till 16 i början av loop L1 ($t1 alltid 8). Flytta initiering av $t0 på rad 5 innan loop L1. T ex:

```
        addi $a1, $zero, 8
        addi $t0, $zero, 4
L1:     addi $a2, $zero, 16
        add $a1, $a1, $t0
        lw $t2, 0($a1)
        add $v0, $v0, $t2
        bne $t2, $zero, L1
        sll $v0, 2
        sw $v0, 4($a2)
```

f. Uppgiften löses genom att studera hur vektorn p är lagrad i minnet och analysera hur vektorelementen accessas. Det visar sig att clear1 har bäst rumslokalitet. clear2 accessar var och en av de N vektorelementen i ordning, vilket är bra, men inom varje element accessas data fram och tillbaks med följande offset från elementets startadress: 0, 12, 4, 16, 8, 20 (om vi antar 4 bytes ordstorlek). Detta medför sämre rumslokalitet än clear1. För clear3 accessas de N vektorelementen i oordning, dessutom accessas data i elementen i oordning varför clear3 har sämst rumslokalitet.