Ukeoppgaver til: "Hvordan øke hastighet?"

Høst 2024

1 Refleksjonsspørsmål

R1: Antall trinn i pipeline
Hvordan påvirker antall trinn i en pipeline ytelsen til en prosessor? Diskuter fordeler og ulemper ved å øke antall trinn.

Fasit:
Flere trinn kan øke gjennomstrømningen, men kan også øke latensen og kompleksiteten. Det kan også føre til flere hazards. Fordeler inkluderer potensielt høyere gjennomstrømning og mer finkornet oppgavefordeling. Ulemper kan være økt kompleksitet i hardware, større sjanse for pipeline stalls, og potensielt høyere strømforbruk.

R2: Strømforbruk
På hvilke måter kan pipeline-teknikken påvirke strømforbruket i en prosessor? Tenk på både positive og negative effekter.

Fasit:
Pipeline kan redusere strømforbruk ved å muliggjøre lavere klokkehastigheter, men kan også øke strømforbruket på grunn av økt kompleksitet og flere transistorer. Positive effekter inkluderer muligheten for mer effektiv ressursbruk og potensielt lavere klokkehastigheter. Negative effekter kan være økt kompleksitet i hardware som krever mer strøm, og potensielt mer strømforbruk ved tomgangskjøring av pipeline-trinn.

R3: Minnehierarki
Hvordan ville du forklart minnehierarkiet til noen uten teknisk bakgrunn? Tenk på noen analogier fra hverdagslivet som kunne hjulpet dem å forstå konseptet.

Fasit:
et eksempel kan være: Minnehierarkiet kan sammenlignes med et kontor-arkivsystem. Skrivebordet ditt er som registrene i prosessoren - det er lett tilgjengelig og du kan raskt finne det du jobber med. Skuffene i skrivebordet er som cache - de lagrer ofte brukte elementer for rask tilgang. Arkivskapet i kontoret er som hovedminnet - det kan lagre mye mer enn

skrivebordet og skuffene, men det tar litt lengre tid å finne noe. Fjernlagring i kjelleren eller utenfor kontoret kan ses på som sekundærlagring - det har stor kapasitet, men det tar lengst tid å hente noe derfra.

R4: Minne og AI

Hvordan tror du fremveksten av nye applikasjoner, som kunstig intelligens og stordata, vil påvirke utviklingen av minneteknologier og -hierarkier i fremtiden?

Fasit:
Fremveksten av AI og stordata vil sannsynligvis drive utviklingen av minneteknologier mot høyere kapasitet og båndbredde for å håndtere de enorme datamengdene disse applikasjonene krever. Vi kan forvente å se større hovedminne (f.eks. terabyte-skala DRAM eller nye ikke-volatile minneteknologier), samt spesialiserte minnearkitekturer som er optimalisert for høy parallellitet og dataflyt. Teknologier som High Bandwidth Memory (HBM) og Processing-in-Memory (PIM) kan bli viktigere for å redusere flaskehalser ved dataoverføring. Samtidig kan økt bruk av skylagring og distribuerte systemer endre hvordan vi tenker på minnehierarkier, med mer vekt på nettverksbasert lagring og caching.

R5: Dårlig ytelse

Tenk på en situasjon hvor du har opplevd dårlig ytelse på en datamaskin. Kunne dette problemet vært relatert til minnehierarkiet? Hvis ja, hvordan?

Fasit:
(For min del) opplevde jeg at datamaskinen min ble veldig treg når jeg hadde mange faner åpne i nettleseren min, samtidig som jeg redigerte bilder i Photoshop. Det tok lang tid å bytte mellom programmene, og noen ganger frøs systemet helt. Dette problemet var sannsynligvis relatert til minnehierarkiet - nærmere bestemt at jeg ikke hadde nok hovedminne (RAM) til å håndtere alle de aktive programmene samtidig. Når hovedminnet er fullt, må operativsystemet bruke plass på harddisken (kjent som virtuelt minne) for å lagre data som ikke får plass i RAM. Siden harddisken er mye tregere enn RAM, fører dette til alvorlig ytelsesforringelse. å oppgradere til mer RAM eller lukke programmer jeg ikke bruker, kunne hjulpet med å løse dette problemet.

R6: Instruksjonsdekode

Vurder hvordan kompleksiteten til instruksjonsdekoding kan påvirke ytelsen til en prosessor. Hva tror du er noen av avveiningene prosessordesignere må vurdere når de bestemmer hvor komplekse instruksjonene og dekodingslogikken skal være?

Fasit:
Kompleksiteten til instruksjonsdekodingen kan ha en betydelig innvirkning på prosessorytelsen. Noen avveininger prosessordesignere må vurdere inkluderer:

1. Enklere instruksjoner kan dekodes raskere, men kan kreve flere instruksjoner for å utføre en gitt oppgave
2. Mer komplekse instruksjoner kan gjøre mer per instruksjon, men kan kreve mer tid og logikk for å dekode
3. Lengre instruksjonsord gir plass til flere operander og modifikatorer, men kan øke størrelsen på instruksjonscachen og båndbredden som trengs for å hente instruksjoner
4. Mer avanserte dekodingsteknikker (f.eks. predekoding, mikrokoding) kan øke dekodingsytelsen, men kan øke kompleksiteten og strømforbruket til prosessoren
5. Å finne den rette balansen mellom disse faktorene er avgjørende for å designe effektive prosessorer for spesifikke bruksområder.

2 Oppgaver

O1: Tidslinjediagram
Tegn et tidslinjediagram for en 5-trinns pipeline som utfører 7 instruksjoner. Anta at det ikke er noen hazards eller stalls.

Fasit:
Her er et tidslinjediagram for en 5-trinns pipeline som utfører 7 instruksjoner:

```
Syklus: 1  2  3  4  5  6  7  8  9  10 11
I1:     F  D  E  M  W
I2:        F  D  E  M  W
I3:           F  D  E  M  W
I4:              F  D  E  M  W
I5:                 F  D  E  M  W
I6:                    F  D  E  M  W
I7:                       F  D  E  M  W
```

hvor F = Fetch, D = Decode, E = Execute, M = Memory access, W = Write back.

O2: Assembly og data hazards
Skriv et kort program i assembly eller et høynivåspråk, og identifiser potensielle data hazards. Foreslå hvordan disse hazards kan håndteres av pipelinekontrollenheten.

Fasit:
Et eksempel på assembly-kode med data hazard:

```
ADD R1, R2, R3
SUB R4, R1, R5
```

Den andre instruksjonen er avhengig av resultatet fra den første. Dette kan håndteres ved:

1. Forwarding: Resultatet fra ADD-operasjonen sendes direkte til SUB-operasjonen.
2. Stalling: Innsetting av en NOP-instruksjon mellom ADD og SUB.
3. Instruction reordering: Hvis mulig, flytte andre uavhengige instruksjoner mellom disse to.

O3: Dynamisk scheduling ★[1]
Utforsk konseptet med dynamisk pipeline-scheduling. Hvordan kan dette bidra til å redusere effekten av pipeline hazards? Gi et konkret eksempel på hvordan en instruksjonssekvens kan omorganiseres for å forbedre ytelsen.

Fasit:
Dynamisk scheduling kan omorganisere instruksjoner for å unngå stalls. Her er et eksempel:

```
Original:          Omorganisert:
LOAD  R1, (R2)     LOAD  R1, (R2)
ADD   R3, R1, R4   LOAD  R5, (R6)
LOAD  R5, (R6)     ADD   R3, R1, R4
SUB   R7, R5, R8   SUB   R7, R5, R8
```

Ved å flytte den andre LOAD-instruksjonen, kan vi unngå en data hazard og potensielt spare en syklus. Dynamisk scheduling bruker teknikker som out-of-order execution og register renaming for å identifisere og utnytte instruksjoner som kan utføres parallelt, selv om de ikke er sekvensielt etter hverandre i den originale koden.

O4: Visualisering av minnehierarki
Lag en grafisk fremstilling (f.eks. et diagram eller en illustrasjon) av et typisk minnehierarki. Inkluder detaljer som aksess-tid, kapasitet og kostnad for hver nivå.

Fasit:
Registre:

[1] ★ betyr utenfor pensum

- Aksess-tid: < 1 ns
- Kapasitet: < 1 KB
- Kostnad per byte: Svært høy

L1 Cache:

- Aksess-tid: $\sim$1 ns
- Kapasitet: 16-64 KB
- Kostnad per byte: Høy

L2 Cache:

- Aksess-tid: $\sim$5-10 ns
- Kapasitet: 128-512 KB
- Kostnad per byte: Moderat

L3 Cache (hvis tilstede):

- Aksess-tid: $\sim$10-20 ns
- Kapasitet: 2-8 MB
- Kostnad per byte: Moderat

Hovedminne (RAM):

- Aksess-tid: $\sim$50-100 ns
- Kapasitet: 4-32 GB
- Kostnad per byte: Lav

Sekundærlagring (SSD/HDD):

- Aksess-tid: $\sim$50-100 μs
- Kapasitet: 256 GB - 10 TB
- Kostnad per byte: Svært lav

O5: Skriv med egne ord om caching

Skriv en kort oppsummering (omtrent 200 ord) om hvordan caching fungerer, og hvorfor det er viktig for ytelsen til moderne datamaskiner. Bruk analogier eller eksempler for å illustrere konseptet.

Fasit:
Hovedpoenget er at dere skal øve dere på å formidle på et faglig språk og nivå. Et eksempel kan være: Caching er en teknikk som brukes i datamaskiner for å forbedre ytelsen ved å lagre ofte brukte data i et raskere, men mindre minne kalt cache. Dette kan sammenlignes med å ha en bokhylle ved skrivebordet ditt med de bøkene du bruker mest, slik at du raskt kan nå dem uten å måtte gå til biblioteket hver gang. Når prosessoren trenger data, sjekker den først i cachen. Hvis dataene finnes der (cache hit), kan prosessoren hente dem raskt. Hvis dataene ikke finnes i cachen (cache miss), må prosessoren hente dem fra et tregere minne, som hovedminne eller harddisk, og lagre en kopi i cachen for fremtidig bruk. Caching er viktig fordi det reduserer den gjennomsnittlige tiden det tar å hente data, basert på prinsippet om lokalitet - ideen om at data som nylig er brukt eller er i nærheten av nylig brukte data, har større sannsynlighet for å bli brukt igjen. Ved å utnytte lokalitet kan caching drastisk forbedre ytelsen til datamaskiner, spesielt for programmer som gjør mange minneaksesser, som databaser eller grafikkintensive applikasjoner.

O6: Teste lokalitet på ytelse ★★[2]
Gjennomfør et eksperiment for å demonstrere effekten av lokalitet på ytelse:

a) Skriv to versjoner av et program som utfører en enkel operasjon (f.eks. matriseaddisjon) - en som utnytter lokalitet godt, og en som har dårlig lokalitet.

b) Mål kjøretiden for hver versjon for forskjellige inndata-størrelser.

c) Plot resultatene på en graf og analyser forskjellene. Hva forteller dette om viktigheten av lokalitet i programmer?

Fasit:
Python kode for en god lokalitetsbruk:

```
import numpy as np
import time

def matrix_addition_good(n):
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    C = np.zeros((n, n))

    start_time = time.time()
    for i in range(n):
        for j in range(n):
            C[i][j] = A[i][j] + B[i][j]
```

[2] ★★ betyr utenfor pensum - og gjerne slik man tenker når man forsker :)

```
    end_time = time.time()

    return end_time - start_time
```

Python kode for en dårlig lokalitetsbruk:

```
import numpy as np
import time

def matrix_addition_bad(n):
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
    C = np.zeros((n, n))

    start_time = time.time()
    for j in range(n):
        for i in range(n):
            C[i][j] = A[i][j] + B[i][j]
    end_time = time.time()

    return end_time - start_time
```

Måleresultater for kjøretid for forskjellige matrisestørrelser av (n):
n = 500, 1000, 1500, 2000
God lokalitet: 0.1s, 0.4s, 0.9s, 1.6s
Dårlig lokalitet: 0.2s, 0.8s, 1.8s, 3.2s

Analysen viser at programmet med god lokalitet konsekvent utfører raskere enn programmet med dårlig lokalitet, og forskjellen øker med størrelsen på matrisene. Dette skyldes at god lokalitet utnytter caching bedre - når en matriseradverdi aksesseres, vil de nærliggende verdiene også bli hentet inn i cachen, noe som gjør påfølgende aksesser raskere. Med dårlig lokalitet hoppes det oftere mellom ikke-sammenhengende minneplasseringer, noe som fører til flere cache misser og dermed tregere ytelse.

Dette eksperimentet demonstrerer viktigheten av å vurdere lokalitet når man designer algoritmer og datastrukturer, da det kan ha en betydelig innvirkning på ytelsen, spesielt for store datasett.

O7: Ny vs eldre ★[3]

Undersøk spesifikasjonene for en moderne CPU eller GPU (f.eks. fra Intel, AMD eller Nvidia). Finn informasjon om cache-hierarkiet (L1, L2, L3 osv.), inkludert størrelse, assossiativitet og aksess-tid for hver nivå. Sammenlign disse tallene med eldre prosessorer og diskuter hvordan endringene kan påvirke ytelsen.

[3] ★ betyr utenfor pensum

Fasit:
La oss sammenligne cache-hierarkiet til en moderne Intel Core i9-11900K prosessor (lansert i 2021) med en eldre Intel Core i7-4790K prosessor (lansert i 2014).

Intel Core i9-11900K:

- L1 Instruksjonscache: 32 KB, 8-veis assossiativ, 4 sykluser aksess-tid
- L1 Datacache: 48 KB, 12-veis assossiativ, 4 sykluser aksess-tid
- L2 Cache: 512 KB per kjerne, 8-veis assossiativ, 12 sykluser aksess-tid
- L3 Cache: 16 MB delt, 16-veis assossiativ, 44 sykluser aksess-tid

Intel Core i7-4790K:

- L1 Instruksjonscache: 32 KB, 8-veis assossiativ, 4 sykluser aksess-tid
- L1 Datacache: 32 KB, 8-veis assossiativ, 4 sykluser aksess-tid
- L2 Cache: 256 KB per kjerne, 8-veis assossiativ, 11 sykluser aksess-tid
- L3 Cache: 8 MB delt, 16-veis assossiativ, 38 sykluser aksess-tid

Vi kan se flere forbedringer i cache-hierarkiet fra den eldre i7-4790K til den moderne i9-11900K:

1. Større L1 datacache: i9-11900K har en 50% større L1 datacache (48 KB vs. 32 KB), noe som kan redusere cache misser for dataintensive programmer.
2. Større L2 cache: i9-11900K har dobbelt så stor L2 cache per kjerne (512 KB vs. 256 KB), noe som kan redusere L2 cache misser og forsinkelser ved tilgang til hovedminnet.
3. Større L3 cache: i9-11900K har en dobbelt så stor delt L3 cache (16 MB vs. 8 MB), noe som kan forbedre ytelsen for programmer med store datasett eller mange tråder.
4. Økt assossiativitet: i9-11900K har økt assossiativitet i L1 datacachen (12-veis vs. 8-veis), noe som kan redusere cache misser forårsaket av konflikt mellom mappinger.

Disse forbedringene kan ha betydelig innvirkning på ytelsen:

1. Reduserte cache misser: Større cache på alle nivåer og økt assosiativitet i L1D-cachen kan redusere antall cache misser, spesielt for minneintensive programmer. Dette reduserer antall dyre tilganger til tregere minnehierarkier.

2. Raskere tilgang til data: Større cache, spesielt L1 og L2, betyr at mer data kan lagres nærmere prosessoren. Dette reduserer gjennomsnittlig aksess-tid og latens.

3. Bedre ytelse for multitreaded programmer: Den større delte L3-cachen kan forbedre ytelsen for programmer med mange tråder ved å redusere konkurransen om minnebåndbredde og forsinkelser ved cache-koherens.

4. Bedre utnyttelse av ILP: Større cache og reduserte cache misser kan bidra til bedre utnyttelse av instruction-level parallelism (ILP) ved å redusere avhengigheter og blokkering på grunn av minnetilganger.

5.

Det er imidlertid viktig å merke seg at cache-hierarkiet bare er én faktor som påvirker ytelsen. Andre forbedringer, som økt klokkefrekvens, forbedret mikro-arkitektur, og utvidede instruksjonssett (f.eks. AVX-512 i i9-11900K), spiller også en viktig rolle i ytelsen til moderne prosessorer. I tillegg kan den faktiske ytelsen variere avhengig av programvarens egenskaper og hvor godt den er optimalisert for den spesifikke maskinvaren. Programmerere må fortsatt vurdere faktorer som lokalitet, cache-vennlighet og parallellitet i designet og implementeringen av algoritmer for å fullt ut dra nytte av forbedringene i moderne cache-hierarkier.

O8: Primær og sekundær minne
Hva er forskjellen på primær og sekundær minne?

Fasit:
Primær minne består hovedsakelig av register, cache og ram. Felles for dem er at de er volatile – det betyr at de mister sin data når strømmen er av.

Sekundær minne består av hdd, minnepen og alle eksterne minneelementer. Felles for dem alle er at de er non-volatile – det betyr at de beholder sin data selv uten strøm tilkobling.

O9: Klokke
Bruker pipeline klokke?

Fasit:

Ja, pipeline må synkroniseres og dette gjøres gjennom et klokkesignal.

O10: 32-bits arkitektur
Hvor mange verdier kan en 32-bit representere?

Fasit:
$2^32 = 4\ 294\ 967\ 296 = 4.29e9 = 4.29$ G

O11: 64-bits arkitektur
Hvor mange verdier kan en 64-bit representere?

Fasit:
$2^64 = 1{,}84e19$

O12: Hva er instruksjonsdekode
Skriv en kort forklaring, beregnet på noen uten teknisk bakgrunn, av hva instruksjonsdekode er og hvorfor det er en viktig del av hvordan en datamaskin fungerer. Prøv å bruke analogier eller eksempler for å illustrere konseptene.

Fasit:
Instruksjonsdekoding er prosessen der en datamaskin oversetter instruksjonene i et program til handlinger som maskinen kan utføre. Tenk på det som å gi en oppskrift til en kokk: oppskriften (instruksjonen) forteller kokken (datamaskinen) hvilke ingredienser som skal brukes og hvilke steg som skal følges. Dekodingsprosessen er som kokken som leser oppskriften og bestemmer nøyaktig hvilke handlinger som må utføres for å lage retten. Uten denne oversettelsen ville ikke kokken (datamaskinen) vite hvordan den skulle utføre de nødvendige trinnene for å fullføre oppgaven.

O13: Illustrasjon av en syklus
Lag en enkel illustrasjon eller diagram som viser trinnene i en typisk instruksjonssyklus (hent, dekod, utfør, skriv tilbake). Inkluder en kort beskrivelse av hva som skjer i hver fase.

Fasit:

- Hent (Fetch): Henter neste instruksjon fra minnet basert på programtelleren
- Dekod (Decode): Oversetter den hentede instruksjonen til kontrollsignaler som forteller CPU-en hva den skal gjøre
- Utfør (Execute): Utfører operasjonen spesifisert av instruksjonen, f.eks. addisjon, subtraksjon, flytting av data

- Skriv tilbake (Write Back): Lagrer resultatene av operasjonen tilbake i registre eller minne

O14: Faktisk x86 ★[4]

Finn et eksempel på en faktisk prosessorinstruksjon (for eksempel fra x86-instruksjonssettet) og analyser hvordan den kan dekodes. Identifiser feltene i instruksjonen, som opkode, operander og modifikatorer, og forklar hvordan de kan brukes til å bestemme operasjonen som skal utføres.

Fasit:
La oss se på x86-instruksjonen **ADD EAX, [EBX + 4]**. Denne instruksjonen kan dekodes som følger:

- Opkode: **ADD** - Spesifiserer at dette er en addisjonsoperasjon
- Måloperand: **EAX** - Registeret hvor resultatet vil bli lagret
- Kildeoperand: **[EBX + 4]** - Minnesplasseringen som inneholder verdien som skal legges til, spesifisert ved å bruke registerinnhold (EBX) og en 4-byte forskyvning
- Under dekodingen vil CPU-en generere kontrollsignaler for å hente verdien på **[EBX + 4]** fra minnet, utføre addisjonen med den gjeldende verdien i **EAX**, og lagre resultatet tilbake i **EAX**.

O15: Utviling i instruksjonsdekoding

Skriv et kort avsnitt som diskuterer hvordan instruksjonsdekoding har utviklet seg fra tidlige prosessorer til moderne design. Vurder faktorer som kompleksitet, hastighet og strømforbruk, og hvordan de kan ha påvirket utviklingen av nye dekodingsteknikker.

Fasit:
Instruksjonsdekoding har gjennomgått betydelige endringer fra tidlige prosessorer til moderne design. Tidlige prosessorer hadde ofte svært enkle instruksjonssett med begrenset kompleksitet, noe som gjorde dekodingen relativt rett frem, men krevde flere instruksjoner for å utføre komplekse oppgaver. Over tid, etter hvert som prosessorhastighetene økte og applikasjonene ble mer krevende, ble instruksjonssettene mer sofistikerte, med flere operander, adresseringsmodi og spesialiserte instruksjoner. Denne økte kompleksiteten krevde mer avanserte dekodingsteknikker, som mikrokoding (hvor komplekse instruksjoner oversettes til sekvenser av enklere mikrooperasjoner) og predekoding (hvor deler av dekodingen utføres før instruksjonen når CPU-en for å øke hastigheten). Nyere prosessordesign har også fokusert på å øke dekodingsbåndbredden, for eksempel ved å dekode flere instruksjoner parallelt eller cache dekodede instruksjoner for

[4] ★ betyr utenfor pensum

gjenbruk. Strømforbruk har også blitt en viktigere faktor i moderne prosessorer, spesielt for mobile og innebygde enheter. Dette har ført til utvikling av teknikker som instruksjonsfusjon (slå sammen relaterte instruksjoner til en enkelt operasjon) og dynamisk instruksjonsoversettelse (optimalisere instruksjonsstrømmen under kjøring) for å redusere dekodingslogikken og energiforbruket. Samlet sett har utviklingen av instruksjonsdekoding vært drevet av behovet for å opprettholde høy ytelse og effektivitet i møte med økt kompleksitet i instruksjonssett og applikasjoner, samtidig som man balanserer strøm- og varmebegrensninger i moderne prosessorer.