**Mandatory Assignment 3: Memory simulator**

**Kort fortalt:**

1. Implementer en cache simulator for et minnesystem i memory.c. Cachen skal ha: et level1 read-only instruction cache, en level 1 data cache og en unified level2 cache. Lvl1 data cachen og lvl2 unified cachen skal begge støtte reads og writes.
2. Bruke simulatoren du lager til å telle cache-hits og cache-misses på benchmarken fra første innlevering. Du skal endre på cache parameterne dine og finne det beste cache designet med den beste ytelsen og med minst mulig bruk av hardwaren.
3. Du har fått en pre-kode som implementerer en API(servitør) mellom et memory subsytem og CPU’en.
4. Din simulator skal utføre instruction fetch, data load og data store operasjoner på en gitt minnesporingsfil som du selv velger.
5. resultatet av programmet/simulatoren er et spor som skal lagres i et binært format som er spesifisert av *p2AddrTr-*strukturen i «byurt.h» filen.
6. Du skal evaluere cachen din ved å lage et memory acess trace som inneholder alle minne-accessene til programet du har valgt. Du bør lage minst to spor:
   1. Et spor som brukes til å teste om simulatoren er korrekt. Antall «cache-hits» og «cache-misses» programmet/sporet skal produsere bør du vite på forhånd.
   2. Et spor av hvor mange minneadresser som blir samlet for referanseindeksen din fra oppgave 1 (eller en annen referanseindeks). Dette skal du bruke til å evaluere cache-parameterne for å finne den beste cache-designen for «benchmarken» din.

**Hele teksten:**

I denne oppgaven skal du implementere en cache-simulator for et minnesystem med et «level 1 read-only instruction cache», en «level 1 data cache» og en «unified level 2-cache». De to sistnevnte skal støtte både «reads» og «writes».

Forkoden gir dere en CPU-simulator som vil utføre minnetilgang mot ditt minnesubsystem. Du skal implementere «memory.c» og det vil ikke være behov for å endre på pre-koden.

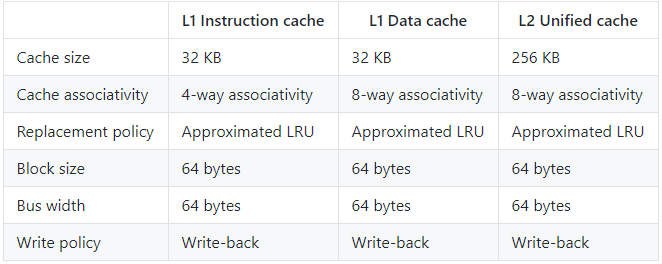
Målet ditt er å bruke simulatoren til å finne den beste cache-designen for «benchmarken» din fra forrige obligatorisk oppgave 1 (eller en annen benchmark). For å gjøre dette skal du måle «cache hit» og «cache miss» rater og eksperimentere med forskjellige cache-parametere.

**Cache-spesifikasjoner**

Noen spesifikasjoner for cachen er fikset: Som et startpunkt bør du bruke følgende spesifikasjoner:

* CPU – «L1 bus width» er 32 bits.
* L1 – «L2 bus width» er L1-blokkstørrelse
* L2 – «RAM bus width» er L2-blokkstørrelsen.

Som et tillegg foreslår tabellen nedenfor disse parametere for hver av cachene. Husk at designet ditt skal gjøre det enkelt å endre de på forskjellige parametrene.



**Cache-simulator implementering**

Som et startpunkt for din implementering av simulatoren, gir vi deg en forhåndskode som implementerer en API (messenger/servitør) mellom et «memory subsystem» og CPU’en. Denne simulatoren utfører henting av instruksjoner(instruction fetch), datainnlasting(data load) og datalagringsminne(data store) operasjoner basert på en minnesporingsfil som du velger. Resultatet (Sporet) lagres i et binært format spesifisert av *p2AddrTr*-strukturen i «byurt.h» filen. Du skal implementere et minnehierarki (lvl1, lvl2 osv…) over *memory subsystemets* *API*, som inneholder moduler for cachene. Det er ikke nødvendig å gjøre noen endringer i pre-koden, bortsett fra memory.c, som er utgangspunktet ditt.

**Cache-utformingen og implementeringen din skal tilfredsstille følgende krav:**

1. Cachene skal være mulig å implementere i en ekte maskinvare; cache-parameterne må være realistiske i henhold til læreboken din.
2. Cache-parametere som f.eks. størrelse, blokkstørrelse, «write-back policy» og tilknytning(associativity) bør lett kunne endres, enten som «runtime» argumenter eller ved bruk av «compile-time flags».
3. L1-data cachen og L2-cachen skal støtte både «reads» og «writes», men L1-instruction cachen må være «read-only» (*det kan være lurt å bruke en generell cache-implementering som du kan bruke for alle tre cachene, også trenger du ikke implementere en mekanisme for å forhindre «writes» til L1-instruction cache, men bare sørg for at du aldri skriver til den*).
4. Cachene bør støtte parameterne og policyene som er spesifisert ovenfor og de som er spesifisert i evalueringsdelen nedenfor.

**Cache-simulator implementering**

Som utgangspunkt for simulatorimplementeringen din, gir vi en forhåndskode som implementerer API-et til et minnesubsystem og en CPU ved hjelp av dette API-et. Simulatoren utfører henting av instruksjoner, datainnlasting og datalagringsminne basert på en gitt minnesporingsfil. Sporet lagres i et binært format spesifisert av p2AddrTr-strukturen i byurt.h-toppfilen. Du må skrive en implementering av et minnehierarki over minneundersystemets API som inneholder moduler for hurtigbufferne. Det er ikke nødvendig å gjøre noen endringer i forhåndskoden, bortsett fra memory.c, som er utgangspunktet ditt.

**Forenklinger**

1. Du kan anta at hver datatilgang er innenfor grensen til en cache-linje.
2. For å simulere «instruction fetch» kan du anta at alle instruksjonene har en fast størrelse (32-bit) og at adressene er ordjusterte (merk at IA-instruksjonene har variabel lengde og kanskje ikke er ordjustert, så du kan ignorere de to siste bitene av minneadressene som er hentet å late som tilgangene er justert).
3. Du trenger ikke å implementere «reads» og «writes» som faktisk overfører data. Bare tell antall «cache-hits» og «cache-misses»

**Metodikk**

Du skal evaluere cachen ved å lage et «memory access trace» som inneholder alle minnetilgangene til programet. Du bør lage minst to spor:

1. Et spor som brukes til å teste om simulatoren er korrekt. For dette bør du vite på forhånd antall «cache-hits» og «cache-misses» programmet/sporet skal produsere, gitt et sett med cache-parametere.
2. Et spor av hvor mange minneadresser som blir samlet for referanseindeksen din fra oppgave 1 (eller en annen referanseindeks). Dette skal du bruke til å evaluere cache-parameterne for å finne den beste cache-designen for «benchmarken» din.

Det kan hende at korrekthetssporingen må opprettes manuelt, startet fra en kjent cache-tilstand og deretter legge til en-etter-en-minne-operasjon i sporet.

For å lage en sporingsfil for benchmarken må du først lage en loggfil over «memory accesses». Deretter konverterer du loggfilen til binært sporingsformat ved å bruke koden *traceconverter.py* fra pre-koden. For å produsere loggfilen kan du enten bruke et minnesporingsverktøy som Valgrind/ Lackey eller ved å analysere programmet ditt for å finne minnetilgangsmønsteret og deretter lage et spor som gjenspeiler dette (for eksempel ved å implementere et skript som lager loggfilen).

**For å kjøre valgrind på benchmarken, kan du utføre følgende kommando:**

*valgrind --log-fil = logfil - verktøy = lackey - trace-mem = ja [programnavn]*

Valgrind vil da produsere en loggfil som kan brukes som input til *traceconverter.py* som vil lage en binær sporings fil kalt «trace.tr».

Du kan deretter kjøre cache-simulatoren med sporingsfilen som input: ./cachesim trace.rt

Pre-koden vil initialisere minneundersystemet ditt ved å kalle på *memory\_init()* og deretter kalle på en av funksjonene for hver minnetilgang i trace.rt:

memory\_fetch() - hvis minnetilgang er en instruksjon

memory\_read() - hvis minnetilgangen er en datalest

memory\_write() - hvis minnetilgangen er en dataskriving

Du bør sørge for at størrelsen på datasettet som ble brukt til å lage sporet er realistisk. I det tilfellet at det ikke er praktisk å lage et minnespor for et realistisk datasett på grunn av tidsbegrensninger eller lagringsstørrelsesbegrensninger, må du diskutere i rapporten hvordan den reduserte størrelsen påvirker de målte resultatene.

**Evaluering**

Målet ditt er å finne den cache-konfigurasjonen som gir benchmarken din den beste ytelsen med lavest mulig kostnad av hardwaren. For å gjøre dette skal du starte med standardparameterne som er oppgitt i Cache-spesifikasjoner delen av oppgaven. Deretter bør du endre en parameter om gangen og måle endringene i cache-hit og cache-miss raten for hver cache. Parameterne du skal endre for hver cache er:

* Total cache size
* Blocksize
* Associativity
* Write-policy (write back, write through)

**Du bør også velge en erstatningspolicy og implementere den:**

* Random
* LRU
* Temporal / spatial

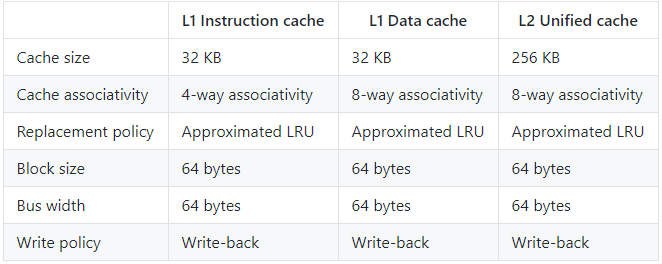
Når du teller cache-hits og cache-misses, bør du skille mellom:

* Layers
* Reads
* Writes

Du har også lov til å endre flere parametere. Merk at antall sett i en gitt cache er en avhengig variabel gitt fra den totale cache-size, block size og associativity(antall måter = antall blokker per sett).

Rapporten din skal presentere resultatene fra målingene dine og gi en diskusjon om kostnadene ved å implementere din endelige cache i maskinvare.

**Hvordan implementere en cache:**

* **Cache**: et lite minne mellom cpu og main memory
* Inneholder i dette tilfellet 2 level
* Hver core i cpu’en har sin egen cache med disse levelene
* Definisjoner på cachene:
* **Cache blocks**: blocks of bytes from main memory that are stores temporarily in the cache memory
  + Block: the main unit of data transfer (typically 64 bytes(512 bits)) between the main memory and the CPU
  + The cache holds a small number of blocks(the frequently used blocks of data)
  + The cpu can request for example 1byte of data from cache, but cannot do this from main memory. Therefore a full block must be transferred from main memory to the cache, so that this 1byte can be transferred from cache to the cpu
  + It is faster to transfer 1 big block of data than many small ones
* **Cache hit**: If a block is in the cache
* **Cache miss**: if a block is not in the cache
* **Bus**: The communication between the CPU and the main memory are done over “busses”. A bus is a set of wires that are carrying data and control signals. One wire can transfers 1bit of data, therefore one wire may be a number of lines transferring chuncks of data. In this example bus width is 64bytes
* **Read request**, the CPU wants to read data from memory:
  + CPU puts a requested address on the bus lines
  + CPU makes its “read request” line high, so it can be written to by memory
  + The memory responds and puts the corresponding data on busses data lines
* **Write request**, the CPU want to write data to memory:
  + The CPU puts an address on the bus lines it also puts the data it wants to write on the busses data lines and it makes its write line high, so it can write to memory
  + The memory sees the address line, data line and it reads that the command is write from the write line, and memory stores the data in its memory
* **NOTE:** The cache sits between the CPU and the main memory, and therefore has access to the same bus lines. The cache is accessed first
* **Temporal locality**: if a byte of memory was used recently, it is likely to be used again soon (applies to both data and instructions)
* **Spatial locality**: If a byte of memory was used recently, nearby bytes are likely to be needed soon (applies to both data and instructions)
* **The working set:** 
  + “the set of bytes that were used recently”
  + OR “the set of bytes that will be used soon”
  + If the working set can be kept in the cache, the program will run faster
  + APPROACH FOR THIS:
    - When we bring bytes into the cache, keep them. Try to keep bytes as long as possible but prefer to keep most recently used bytes.
    - When we bring bytes into the cache, also bring in the nearby bytes AKA blocks (spatial locality)
* **When the CPU wants to write to cache:**
  + **Write-hit**, the block that is going to be modified is in the cache:
    - Write-through: cache immediately writes modified block back to memory
    - Write-back: the cache waits and writes block to memory when it is evicted(something else must take its place)
    - **Note:** cache needs a “dirty bit”. If bit is 1 it means that the block has been modified, if it is 0 it means it is not modified
  + **Write-miss**, the block that is going to be modified in not in the cache:
    - Write-allocate: cache will read and copy the block(which is going to be updated) from memory. The copied block in the cache is then going to be updated
    - Write-no-allocate: CPU sends the write straight to memory. It does not copy block into cache first.
* **Block addresses:**
  + The address is divided into three parts:
    - Block offset: the lowest bits that tells us where we are in the block
    - Index: tells us where in a cache a block can be located
    - Tag: tells us which block from memory we have in the cache
      * **Example:** 2-way associative cache: 00101001

The two lowest 001010 ”01” = offset

Next two lowest 0010”10”00 = index

4 highest bits “0010” 1000 = tag

* + - **How to calculate offset, index and tag:**
      * Offset = log2(block size) OR 2^n = block size
        + Log2(64) = 6 bits for the block offset
        + 2^6 = 64, AKA 6bits is needed
      * Block Offset = Memory Address mod 2^n
      * Block Address = Memory Address / 2^n
      * Set Index = Block Address mod 2^s
      * Number of sets = cache size / (way-assosiative \* block size)
        + 32Kilobit / (4 \* 64 bytes) = 4096 bytes / (4\*64) = 16 sets
      * Index = log2(cache size / block size)
        + Log2(4096bytes / 64 bytes) = 6 bits
      * Tag = the remaining bits of the address when both offset and index are taken
        + 32 – 6 – 6 = 20 bits
* **Set associative cach:**
  + Basicly a cache that is divided into different sets, and each set is then divided into a given number of parts.
  + A 4-way associativity cache means that a cache divided into sets, and each set is then divided into 4 parts. Same as with 8-way associative is divided into 8 parts
  + How to figure out how many sets there are in a cache:
    - 32byte cache, 2-way associative, 4byte cache line
    - Set-size = 4byte cacheline \* 2way-associative = 8byte for each set
    - 32byte cache / 8 byte for each cache = 4 total sets
    - 64 byte cache, 4-way,
* 1KiloBit = 1000 bits
* 1byte = 8 bits
* 1 kilobit = 125bytes
* 32 kilobit = 4000 bytes = 32 000 bits
* 4000 bytes / 64 bytes =